ON THE DESIGN OF PROGRAM EXECUTION ENVIRONMENTS FOR NON-SIGHTED
COMPUTER PROGRAMMERS

By

ANDREAS MIKAL STEFIK

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2008

© Copyright by ANDREAS MIKAL STEFIK, 2008
All rights reserved
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of ANDREAS MIKAL STEFIK find it satisfactory and recommend that it be accepted.

______________________________
Chair
ACKNOWLEDGEMENT

Thanks especially to Melissa, my perpetual participant and sounding board, and my graduate committee. Thanks to Chris, whose word smithing talents helped me learn how to talk about my work. Thanks to Bob, who helped me understand that basic research should inform a research agenda and influence a design, and thanks to Fitz, who encouraged the idea that audio could be useful for programming a computer. Also, thanks to Jon Brown, who helped formalize part of my study designs.

I would like to thank Behrooz Shirazi, Roger Alexander, and the WSU graduate and professional student association for helping to provide funding for the numerous conferences I have been able to attend, and general funding I have received, as a graduate student. Special thanks to Shirazi for providing additional funds for conducting a double blind empirical study.

I would also like to thank Ed Gellenbeck, James Schwing, Razvan Andonie, Grant Eastman, Diana Rae Springer-Lund, John Durham, and Katrina Durham for their help setting up and conducting the empirical study at Central Washington University. Special thanks goes to Ed Gellenbeck and John Durham, as the work at Central would not have been possible without their contribution.

No science on how humans use computers can be conducted unless individuals are willing to endure an empirical study. I would like to thank my participants, including many members of WSU’s Electronic Music Association, and students both at Central Washington University and Washington State University, some of whom have endured — literally — years of poking and prodding. Last, I would like to thank everyone on the blind programmers mailing list at www.blindprogramming.com. Your comments and suggestions have deeply inspired me.
Comprehending and debugging computer programs is difficult for sighted programmers. For the non-sighted, these tasks are even more difficult. Since non-sighted computer programmers have inherent physiological limitations, they must rely on auditory cues to represent the computer programs they are writing, modifying, exploring, or debugging. State-of-the-art interfaces for non-sighted programmers work by retrofitting screen readers onto existing development environments. As such, these environments are built with sighted usability in mind, but often pay little heed to the needs of a community that cannot see the computer screen.

In contrast, this research develops an approach to building tools for the non-sighted computer programmer from the ground up. The core of this ground up approach lies in the use of basic research and empirical studies to inform auditory cue design independently of visual environments. Basic research can, for example, tell us how well auditory cues can be understood, under what context they should be used, and when working memory is being over-taxed—a typical problem with non-sighted interfaces in general.

In this dissertation, I present a line of research related to non-sighted computer programmers. The primary contributions include a characterization of the design space of audio-based program
execution environments, a novel empirical paradigm for analyzing the comprehension of auditory cues (artifact encoding), a research tool for exploring that design space (the Sonified Omniscient Debugger), and empirical evidence that building audio-based programming tools for the non-sighted from the ground up is superior to retrofits of existing visual tools.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Visual Features of the Modern Integrated Development Environment</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Where am I?</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 What’s up?</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Research Questions</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1 What is the design space of auditory cues used for programming a computer?</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 How do we measure auditory cue comprehension?</td>
<td>7</td>
</tr>
<tr>
<td>1.3.3 How can we develop an audio-based program execution environment that is more effective than the present state-of-the-art?</td>
<td>8</td>
</tr>
<tr>
<td>1.4 The Thesis</td>
<td>8</td>
</tr>
<tr>
<td>1.4.1 Design Space of Non-sighted Programming Environments</td>
<td>9</td>
</tr>
<tr>
<td>1.4.2 Artifact Encoding - Empirical Study Paradigm</td>
<td>12</td>
</tr>
<tr>
<td>1.4.3 Experiment 1 - Auditory Cue Comprehension</td>
<td>14</td>
</tr>
<tr>
<td>1.4.4 Program Execution and Debugging Environment</td>
<td>15</td>
</tr>
</tbody>
</table>
4. EXPERIMENTAL PARADIGM: ARTIFACT ENCODING . . . . . . . . . . . . . . 46
4.1 Previous Empirical Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
   4.1.1 Multiple Choice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
   4.1.2 Usability Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
   4.1.3 Software Engineering Metrics . . . . . . . . . . . . . . . . . . . . . . . . 50
   4.1.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.2 Artifact Coding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
   4.2.1 Example: Animal Identification . . . . . . . . . . . . . . . . . . . . . . . 52
   4.2.2 Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
   4.2.3 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
   4.2.4 Coding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
   4.2.5 Measures and Grading . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

5. EMPIRICAL STUDY: AUDITORY CUES FOR CONVEYING SCOPE . . . . . . . . . 68
5.1 Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   5.1.1 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
   5.1.2 Participants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
   5.1.3 Materials and Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
   5.1.4 Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
   5.2.1 Reliability Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
   5.2.2 Performance Measures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
   5.2.3 Survey Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
   5.3.1 Qualitative Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.3.2 Threats to Validity .................................................. 91
5.4 Conclusion ............................................................... 91

6. DESIGNING AN AUDIO BASED PROGRAM EXECUTION ENVIRONMENT . 93
6.1 SOD: The Sonified Omniscient Debugger ................................. 93
   6.1.1 Visual User Interface .............................................. 94
   6.1.2 Auditory User Interface .......................................... 95
   6.1.3 Discussion .......................................................... 103
6.2 Architecture ............................................................ 103
6.3 Conclusion ............................................................... 107

7. EMPIRICAL STUDY: SCREEN READER VS. SOD ....................... 109
7.1 Sample Session with SOD ............................................... 110
7.2 Methods ................................................................. 113
   7.2.1 Design ............................................................... 113
   7.2.2 Participants ........................................................ 114
   7.2.3 Materials and Tasks .............................................. 114
   7.2.4 Procedure .......................................................... 115
7.3 Results ................................................................. 116
   7.3.1 Performance Measures ............................................ 117
   7.3.2 Survey Results .................................................... 118
7.4 Discussion .............................................................. 120
   7.4.1 Theoretical Explanation ......................................... 120
   7.4.2 Threats to Validity ............................................... 123
7.5 Conclusion .............................................................. 124
8. CONCLUDING REMARKS ......................................................... 125

8.1 Contributions ................................................................. 126
  8.1.1 The Design Space of Auditory Cues ............................... 126
  8.1.2 Experimental Paradigm: Artifact Encoding ...................... 126
  8.1.3 Empirical Study: Auditory Cues for Conveying Scope ........ 126
  8.1.4 Designing an Audio Based Program Execution Environment .... 127
  8.1.5 Empirical Study: Screen reader vs. SOD ....................... 127

8.2 Future Research .............................................................. 127
  8.2.1 Artifact Encoding Automation ...................................... 128
  8.2.2 Non-sighted Navigation ............................................. 128
  8.2.3 Natural Non-sighted Programming ................................. 129
  8.2.4 Visual vs. Visual + Audio ......................................... 130
  8.2.5 Expanding SOD ..................................................... 130

8.3 Conclusion ................................................................. 131

APPENDICES

A. THE ARTIFACT ENCODING TRAINING MANUAL ....................... 133
  A.1 Simple Participant answers .......................................... 133
  A.2 Complex Participant Answers ....................................... 135

B. SOD AUDIO INTERFACE ................................................... 145
  B.1 Visual Studio 2005 + Jaws 9 ........................................... 145
    B.1.1 Characters ........................................................ 145
    B.1.2 Arrow Keys ..................................................... 146
    B.1.3 Special Arrow Key Functions .................................. 146
    B.1.4 Local variable window ......................................... 147
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1.5 Navigation</td>
<td>147</td>
</tr>
<tr>
<td>B.2 Sonified Omniscient Debugger</td>
<td>148</td>
</tr>
<tr>
<td>B.2.1 Characters</td>
<td>148</td>
</tr>
<tr>
<td>B.2.2 Arrow Keys</td>
<td>149</td>
</tr>
<tr>
<td>B.2.3 Special Arrow Key Functions</td>
<td>149</td>
</tr>
<tr>
<td>B.2.4 Local variable window</td>
<td>149</td>
</tr>
<tr>
<td>B.2.5 Navigation</td>
<td>150</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>A table with multiple choice options that may identify the code represented in Figure 4.1.</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>A table of different sounds used in formal experiments, their intended meanings, and how they are coded.</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>An example comparison of a participant’s answer (Left) and the correct answer (right). The total comprehension score for this example is 4 out of 6, or 66.6%.</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>This table shows the number of participants in each cell of the 2x2 design.</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>This table shows the number of students that participated at each time of the study.</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>This table shows which group was assigned at which time for the experiment.</td>
<td>73</td>
</tr>
<tr>
<td>5.4</td>
<td>A summary table of the experimental results ($^* = \alpha &lt; 0.05, ^{**} = \alpha &lt; .01$).</td>
<td>78</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Screenshot of the Netbeans 5.5 Integrated Development Environment (IDE) while debugging source code.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>This figure represents the situation a non-sighted programmer is in when programming. The current line of code can be read by a screenreader, which represents only static information about a computer program. Since only the current line can be read, the non-sighted programmer must, somehow, determine the current context under which they are working. This inevitably leads to questions like, “Where am I” and “What’s up?”</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>As opposed to retrofitting visual technology for the non-sighted, this thesis focuses on building non-sighted programming environments from the ground up. This ground up approach relies on both basic research studies (Chapter 5) and applied research studies (Chapter 7) to inform the design.</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>This nested IF statement would be sonified as, “if true, 1 nested if true.”</td>
<td>10</td>
</tr>
<tr>
<td>1.5</td>
<td>Participants in pilots often misinterpreted the scoping level of program constructs, like those in Figure 1.4, with what is shown here if scoping level is not explicitly sonified.</td>
<td>11</td>
</tr>
<tr>
<td>1.6</td>
<td>Given a = 0, b = 1, c = 1, and d = 2 the text would be, “if false, 1 nested loop iteration 1, 1 nested loop iteration 2, end loop.”</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>A basic overview of Baddeley’s working memory model taken from Baddeley (2002). Areas that are grayed, termed crystallized in the figure, interact with long term memory.</td>
<td>33</td>
</tr>
</tbody>
</table>
3.1 A summary of the types of auditory cues described in this thesis.  37

3.2 The top part of this figure shows modern visual tools and techniques for programming, from left to right UML, the Netbeans Editor, and the Netbeans debugger. The bottom represents the corresponding design space of auditory cues for the non-sighted programmer, which uses audio to replace visual stimuli.  39

3.3 A more detailed view of execution related auditory display information.  42

3.4 A typical recursive function for computing a factorial.  43

3.5 A simple if statement which, at runtime, can have a number of auditory cues associated with it.  45

4.1 An if statement  47

4.2 If a participant wrote A, B, C, or D when choosing amongst 3 similar examples, there is no way to know what part of an auditory cue actually aided comprehension.  48

4.3 This shows the notation participants use to represent a true or a false if statement.  55

4.4 This notation shows what “if ( F )” really means in practice, that the code executed into an else block.  56

4.5 In this example, the 2 indicates the number of iterations of the loop. On the inside of the loop, the first T indicates what happened on the first iteration, while F indicates what happened on the second iteration.  56

4.6 In this more complex example of a loop, there are actually two if statements. The first if statement evaluated to true on the first iteration and false on the second. The second if statement did the opposite.  56

4.7 This participant’s answer would be coded as ITNIFU.  58

4.8 This is an example of code where a loop exists, but never actually executes. This would be correctly coded as a single L.  58

4.9 This is the auditory cue that would be heard for a loop executing four times.  58
4.10 The coding for this answer is be LNU LNU LNU LNU. .......................... 59
4.11 This is the auditory cue for a more complex loop, which iterates twice, each time
with different behavior. ................................................................. 59
4.12 The coding for this answer is be LN ITNIFU U LN IFNITU U. Spaces were added
between iterations of a loop to make this coding easier to read. .................... 60
4.13 This example is the same as Figure 4.12, although it is represented differently to
show where control flow goes in the program more easily. The underscores here
represent regions of the code that control flow does not reach. .................... 60
4.14 An example where the Z character is used in coding. The coding here would be
ITZIT. ............................................................................................. 60
4.15 A basic overview of Artifact Encoding, including the automatic grading process. . 61
4.16 A Screenshot of the artifact encoding automatic grading program, using a modifi-
cation of the Needleman-Wunsch algorithm Needleman & Wunsch (1970). ...... 63
4.17 On the left is the actual answer for a particular problem, whereas on the right is
a participant’s answer. If cascading errors were not taken into account, the par-
ticipant would get two points docked for scoping errors, while with the cascading
errors correction they would be docked one. ........................................ 65

5.1 This code generated the auditory cues for task 6 in the experiment. ............... 72
5.2 An example from the passive listening portion of the training during this study.
The caption in the study states: “The sounds in this example are the same as those
before, but in this case, an IF statement is nested within a while loop. This loop
executes twice. During the first iteration of the loop the IF statement evaluates to
TRUE and during the second iteration it evaluates to FALSE. The final sound pro-
duced by this execution sequence is LOOP ITERATION 1, 1 NESTED IF TRUE,
LOOP ITERATION 2, 1 NESTED IF FALSE, END LOOP.” ....................... 75
5.3 This is a simple example of a nested if statement. The caption in the study states:
“The auditory cue, assuming both statements evaluated to true, would be IF TRUE,
1 NESTED IF TRUE for the scoping group, while for the no scoping group it
would be IF TRUE IF TRUE.”

5.4 This example is from the answer preparation phase of the experiment. The caption
in the study states: “The number two in the while loop means the body of the loop
executed twice, while the letters T,F indicate that the first time the loop executed
the if statement evaluated to true, but that it evaluated to false the second time.”

5.5 This figure shows the percent correct for the aggregate artifact encoding score,
called comprehension, on the y-axis. On the x-axis is the independent variable of
scoping cues, either without them (left), or with them (right).

5.6 This figure shows the percent correct for the aggregate artifact encoding score,
re comprehension, on the y-axis. On the x-axis is the independent variable of experi-
eence, whether the previous number of programming classes participants had taken
was either 0 (left), or 1 or more (right).

5.7 This figure shows the number of missing loops and if statements, summed from
all ten tasks, on the y-axis. On the x-axis is the independent variable of scoping
cues, either without them (left), or with them (right).

5.8 This figure shows the number of missing loops and if statements, summed from
all ten tasks, on the y-axis. On the x-axis is the independent variable of experience,
whether the previous number of programming classes participants had taken
was either 0 (left), or 1 or more (right).

5.9 This figure shows the percent of program constructs, loops or if statements, that
were placed at the correct scoping level. On the x-axis is the independent variable
of scoping cues, either without them (left), or with them (right).
5.10 This figure shows the percent of program constructs, loops or if statements, that were placed at the correct scoping level. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

5.11 This figure shows the number of flipped booleans in participants’ answers. On the x-axis is the independent variable of scoping cues, either without them (left), or with them (right).

5.12 This figure shows the number of flipped booleans in participants’ answers. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

5.13 This figure shows the percent correct for the comprehension metric for every task.

5.14 This figure shows the percent correct for scoping in each task.

5.15 This figure shows the percent of correctly placed program constructs at each level of scope.

5.16 An example auditory cue representing a loop that executes twice and has no significant control flow within it.

5.17 The correct written answer for the auditory cue listed in Figure 5.16.

5.18 A relatively uncommon misinterpretation of the auditory cue listed in Figure 5.16. This mistake was usually made by those registered for their first computer programming course.

5.19 Some participants misinterpreted the auditory cue 3 nested if true to mean three if statements that executed, all with the same truth value.

6.1 Screenshot of the SOD user interface.
6.2 Assuming variables a and b were equal, this statement would be read, “if true.” The compiler automatically determines that the value of \( a == b \) should be spoken as either true or false when in the context of an if statement.

6.3 Assuming variables a and b were equal, this statement would be read, “set c to 1” The compiler in this case does not assume that the value of 1 is supposed to mean either true or false.

6.4 A set of nested if statements.

6.5 An incorrect interpretation of “3 nested if true.”

6.6 A more complex set of nested if statements. One auditory cue related to scoping might indicate scoping depth, like “1 nested if true,” while another might indicate the beginning and ending of the if statements using the words “nest,” “unnest,” or something similar.

6.7 A loop

6.8 A nested loop

6.9 A simple function, myFunc, being called from a main method in C.

6.10 A function that is called as part of an expression will be aurally hidden, telling the user the value of the expression but not indicating that a function was called. In this case, the line, \( \text{result} = \text{myFunc}(); \) would be sonified as, “set variable result to 5.”

6.11 Overview of the SOD architecture

7.1 Snapshot of SOD interface while performing “replace values” comprehension and debugging tasks

7.2 Total time on task by condition

7.3 Comprehension and debugging performance by condition

7.4 Exit questionnaire results by condition
Dedication

To Melissa, my beloved guinea pig.
CHAPTER 1

INTRODUCTION

When computer programmers who possess sight analyze a piece of software, they take advantage of rich visual cues. Such cues aid in comprehension of both program constructs (e.g., if or while), and relationships between program constructs (e.g., using tabbed based code formatting to represent scoping relationships). For non-sighted computer programmers, however, visual cues are not available, requiring alternative methods for representing the same, or additional, information about the computer code. One obvious choice is the use of auditory cues.

The purpose of my work is both to develop auditory tools to help non-sighted computer programmers understand computer programs, and to empirically test these tools in studies involving humans. This thesis makes five primary contributions. First, I provide a map of the design space of programming environments that use audio instead of visual stimuli. While visual environments for programming have been well explored, the design choices that must be made in the development of an environment for non-sighted programmers is neither obvious nor discussed in the literature.

The success of any audio based interface hinges on its ability to make the meaning of the auditory cues obvious to users. I have created a technique I call Artifact Encoding, which uses a series of complex algorithms and procedures to measure the human comprehension of auditory cues used for computer programming.

The third contribution is an empirical study on the comprehension of behavioral runtime cues—a type of auditory cue that is output by the debugger. Specifically, I test whether a particular form of scoping cue aids the programmer in comprehending a computer program.

The fourth contribution is a tool I built for non-sighted computer programmers that allows me to explore one segment of this design space. Specifically, I have created a C-interpreter and a program execution environment. This environment integrates several phases of the modern compiler into speech based output, which allows for a richer and more informative audio based user interface.
Last, in a summative study, I compare my program execution environment to both a modern screen reader and a visual programming environment using realistic programming tasks. This experiment helps indicate whether my ground up approach transfers to a realistic setting.

To get a better idea of the types of issues involved in this research, I think it is beneficial first to analyze the types of visual cues sighted programmers take for granted. Then, consider the specific problems addressed in this thesis and then the thesis itself.

1.1 Visual Features of the Modern Integrated Development Environment

Modern programming environments include a series of tools and techniques to help the programmer with daily development tasks. Some tools allow the developer to navigate computer code, or to see individual pieces of code graphically; others highlight the currently executing line of code, graphically “guard” code, or indicate information about a piece of syntax through syntax highlighting. None of these features exists for the non-sighted programmer. What types of auditory cues would be most beneficial for this community goes relatively unexplored. In this section, I highlight the features that sighted programmers take for granted, focusing predominately on runtime features.

Consider the visual features in Netbeans 5.5, an integrated development environment (IDE) typically used for the Java programming language. Netbeans has a rich visual interface, including visual representations of programming projects, files, and runtime attributes. It includes tools with names such as inspector, navigator, or palette. Virtually any modern environment has a way to visually present source code, typically with popup code completion and syntax highlighting. Each of these visual features was designed to help the developer with some form of cognitive task, like comprehending an attribute of the source code, reducing working memory constraints, or visually building up a graphical application. Since this thesis is primarily concerned with enabling non-sighted programmers to comprehend the run-time behavior of programs, I focus only on the runtime components in the Netbeans IDE for the rest of this section.
Consider Figure 1.1, in which Netbeans is debugging an application. Note that in the middle of the screen is a section of code labeled `private void buildMenuItemActionPerformed(java.awt.event.ActionEvent evt)`. This method represents the action that is taken when a particular menu item is executed, which causes the method `build()` to be called.

For non-sighted programmers to have an easy-to-use interface, they need good auditory cues to replace the visual cues in modern environments. In Figure 1.1, notice there is green highlighting on the line of code labeled `build()`, which indicates the current point of execution. Sighted programmers can easily glance up or down from this green line to understand the context under which the program is executing. Techniques for taking an audio glance (Stevens, 1996) of an executing program, however, have not yet been developed.

---

1The blue lines indicate guarded code, which means the code cannot be modified by the user.
To garner state information from the development environment, a user can use the local variables and watch windows as shown on the bottom of Figure 1.1. The local variables window allows the user to navigate through the current state of the program. The watch window allows the user to type in a particular command or variable, whose value the user would like to track. For non-sighted programmers to use a tool like this, they would first need to memorize their current location in the source, and the open file, then attempt to navigate (Smith et al., 2004) to the appropriate node. Visual users, on the other hand, see near instantaneously (through glancing) where the green line is in the program, as well as the current state of the program.

The elements included in modern IDEs for programming a computer are sophisticated, rich, informative, and almost exclusively visual. While it seems likely that visual interfaces will help sighted users more than purely audio interfaces, even the possibility of using audio has gone relatively unexplored in the literature. Developers of computer code, in the best case, connect their code to an accessibility API, which allows screen readers like JAWS \textcopyright 9 (Freedom-Scientific, 2008) to interact with it. However, screen readers do what their name implies: they read what is on the screen. They are not designed for the highly complex nature of computer programming.

Keeping in mind what type of visual features modern development environments make available, I now turn to the primary problems presented in this thesis.

1.2 Problem Statement

Non-sighted computer programmers face significant challenges. I have identified two primary problems non-sighted programmers face when trying to understand the execution of a program. I name these two problems “Where am I,” and “What’s up?” Figure 1.2 summarizes the types of questions a non-sighted programmer might ask themselves during the execution of a program.

1.2.1 Where am I?

When sighted users look at a section of source code, they can use visual cues to determine their textual location, or if debugging, their textual location and their location in the current execution
Figure 1.2: This figure represents the situation a non-sighted programmer is in when programming. The current line of code can be read by a screenreader, which represents only static information about a computer program. Since only the current line can be read, the non-sighted programmer must, somehow, determine the current context under which they are working. This inevitably leads to questions like, “Where am I” and “What’s up?”
of a program. Our ability, as humans, to remember information or retrieve sets of concepts is dependent upon the conditions under which we encode that information and the the conditions under which we retrieve it. This theory is known as the encoding specificity principle (Thomson & Tulving, 1970). Non-sighted programmers, in contrast, must encode information about a computer program using only auditory display techniques.

Consider then, how non-sighted programmers must encode information about their current textual, or execution time, location in a program. First, programmers must listen to the code, line by line, until they get to a certain location. Once at that location, they must remember as much as possible about previous code and the relationships between constructs in the code, in order to understand the context in which they are working.

1.2.2 What’s up?

In order for non-sighted computer programmers to be able to effectively debug a computer program, they need to be able to determine the state of that program at any arbitrary point along its execution. Good auditory cues should also help by indicating where that state occurred in the program’s execution.

For non-sighted programmers, listening to the execution of a debugger is like listening to a long sentence. The context under which humans hear a sentence can have significant effects on how they interpret it (Miller et al., 1951). Meaningful sentences can help humans predict the last word of the sentence (Tyler & Wessels, 1983). Therefore, auditory cues representing the state of a program should allow a programmer to anticipate what is happening in the program as best as possible.

Consider the following line of code a screen reader might read to a human when running a computer program: “if left parenthesis a equals equals b ampersand ampersand c equals equals d right parenthesis.” If non-sighted computer programmers hear a typical phrase like this, how can they determine the state of a computer program? Clearly, in this context they cannot, as this
information gives only static cues, not dynamic cues. Further, even if the previous phrase contained dynamic cues, would they be effective cues for determining the state of the program?

In general, what types of auditory cues are effective for determining the state of a computer program? Good cues for determining the state of a computer program, that tell the programmer, “What’s up” should maximize a human’s limited working memory, should give a clear indication of temporal location in an execution sequence, and should be as short as possible. Since the mind’s phonological loop can store only a limited few seconds of audio (Baddeley, 1992), all else being equal, the shorter the auditory cues the better. The problem of how to indicate state to a non-sighted programmer goes virtually unexplored in the contemporary literature.

1.3 Research Questions

The problems identified above give rise to the following research questions:

1.3.1 What is the design space of auditory cues used for programming a computer?

Non-sighted computer programs use auditory cues to acquire information from the computer. However, not all cues are useful in every context. For example, while designing the high level structure of a computer program, analogously to a sighted user creating UML, non-sighted programmers need cues that indicate high level abstractions of the source code. In contrast, while editing source code, or running code in a debugger, significantly different kinds of information are needed.

1.3.2 How do we measure auditory cue comprehension?

Calculating how well a human can comprehend auditory cues is a significant challenge. Cues can be remembered, or understood, differently depending upon the context (Thomson & Tulving, 1970; Miller et al., 1951). Different programming languages can define new types of context, as in the aspect oriented community. Non-sighted programmers need auditory cues that maximize their understanding of a computer program.
1.3.3 How can we develop an audio-based program execution environment that is more effective than the present state-of-the-art?

Non-sighted computer programmers attempt to learn to program by using modern development environments (e.g., Visual Studio, Eclipse, or Netbeans), that have been retrofitted with a screen reader (e.g., JAWS or Window Eyes). However, these environments are designed with visualization in mind, not auditory display. This makes sense because “most programmers” are sighted. For the non-sighted, however, the visual nature of an environment is a significant hurdle to overcome. Navigating between windows is possible, but how navigation occurs is application-dependent. Debugging is possible using command line debuggers, for example by using print statements and having the screen reader read them, but techniques like these are archaic and tedious.

1.4 The Thesis

Comprehending and debugging computer programs are inherently difficult tasks for sighted programmers. These tasks are even more difficult for non-sighted programmers, who must rely exclusively on audio-based representations of programs. The current state-of-the-art approach to building program execution and debugging environments for non-sighted programmers is to retrofit existing visual environments with screen readers. Because of intrinsic differences in the way humans process audio (serially) and visual information (in parallel), I argue that effective programming technologies for the non-sighted must instead be built from the ground up, using both basic and applied research studies. This ground up approach is summarized in Figure 1.3.

In order to design a program execution environment for non-sighted programmers, I propose to conduct a series of empirical studies on sighted proxies. The non-sighted programmer community is relatively small, geographically dispersed, and relatively difficult to study directly. Research on non-sighted programmers is rare in the literature due to a lack of available participants, yet the needs of the community, due to their disability, are great. Using sighted proxies, we can obtain a baseline for increasing the usability of programming tools as a first step toward helping the
community. I now provide a brief summary of the dissertation.

1.4.1 Design Space of Non-sighted Programming Environments

While my work focuses predominately on runtime tasks, full scale commercial environments would need auditory cues for non-runtime tasks as well. Unfortunately, since non-sighted programming environments have received almost no attention in the literature, this design space goes relatively unexplored. In Chapter 3, I present a broad overview of this design space, while again focusing on the most significant component of my research, runtime auditory cue design. That chapter is related to the research questions, “What is the design space of auditory cues used for programming a computer?”

It is important, when doing this type of work, to realize that different auditory cues are useful in different circumstances. Screen readers read syntax related cues. Syntax cues (e.g., a equals asterisk b asterisk c semicolon) are easy to edit, but difficult to interpret. Semantics cues (e.g., a is set to dereferenced pointer b multiplied by c) more obviously state the meaning of a phrase, but make it harder to edit the raw syntax. At runtime, auditory cues like “set a to 5” tell the programmer
int main() {
    int a = 5, b = 5, c = 5, k = 0;
    if(a==b) {
        k++;
        if(c == b) {
            k++;
        } else {
            k--;
        }
    } else {
        k--;
    }
    return 0;
}

Figure 1.4: This nested IF statement would be sonified as, “if true, 1 nested if true.”

the behavior of a program, but tell little about its syntax.

In the Sonified Omniscient Debugger (SOD), where I focus predominately on runtime auditory cues, select program constructs are first class entities. One might initially assume that a mapping from speech based audio to programming constructs is trivial, perhaps using only the name of the program construct. Unfortunately, this simple approach has been ineffective in pilots. Instead, audio tools seem to benefit from, 1) a static analysis of the code, 2) supplementing programming constructs with “code features” and 3) ensuring that names do not overlap, like the number four and the programming concept of a for loop (Begel & Graham, 2005; Stefik et al., 2006a).

Consider the programming example given in Figure 1.4. Note that in this code, the first if statement will evaluate to true, and then a nested if statement will similarly evaluate to true. In my speech based system, when the debugger reaches the line, if(a==b) { it will sonify the construct by pausing the debugger and saying, “if true.” This cue states the behavior of the if statement, but if users do not expect the statement to evaluate to true, they may need to request a syntax or semantics cue. Next, on the line k++ the debugger will say, “set k to 1” indicating the new value of k after being incremented.

The next line I will sonify is, if(c == b) {, which is sonified as “1 nested if true.” If,
```c
int main() {
    int a = 5, b = 5, c = 5, k = 0;
    if(a==b) {
        k++;
    }
    if(c == b) {
        k++;
    }
    else {
        k--;
    }
    return 0;
}
```

**Figure 1.5:** Participants in pilots often misinterpreted the scoping level of program constructs, like those in Figure 1.4, with what is shown here if scoping level is not explicitly sonified.

Instead, this statement is sonified as “if true,” participants have difficulty determining the scope of the program construct. Figure 1.5 gives an example of the typical mistake made by a participant if the auditory cue used is, “if true” instead of “1 nested if true.” I test this hypothesis formally in Chapter 5. The point is that well designed auditory cues should give the non-sighted programmer the context under which a construct is executing.

Loops are presented in a similar fashion. Figure 1.6 shows an example of code which includes a loop. In this case, I indicate scoping to avoid errors like those in Figure 1.5. Since my tool is a program execution environment, it needs to aurally indicate iterations of loops. One possibility is to use the equivalent of an audio progress bar to indicate estimated progress through a loop. However, due to the halting problem, progress can be detected only in special cases, and thus I decided to indicate how many times the loop has iterated. At first, the iteration number was a zero based index, but the most recent feasibility study (Stefik et al., 2007) appeared to show “off by one” errors in the number of loop iterations, and as such, I have moved to a one based index. Thus, the first iteration of the loop will say the scoping level of the construct (e.g., 1 nesting), the construct name, “loop,” and the loop iteration number. Figure 1.6 gives an example of an auditory
int test(int a, int b, int c, int d) {
    int k = 0, i = 0;
    if(a == b) {
        k++;
    }
    else {
        k--;
        while(i < d) {
            k++;
            i++;
        }
    }
    return k;
}

Figure 1.6: Given a = 0, b = 1, c = 1, and d = 2 the text would be, “if false, 1 nested loop iteration 1, 1 nested loop iteration 2, end loop.”

cue which includes a loop.

1.4.2 Artifact Encoding - Empirical Study Paradigm

Current auditory display research makes broad claims, like “... [earcons] are shown to be significantly less accurate and significantly more mentally challenging” (see e.g., Finlayson & Mellish, 2005, pg. 132). While this result is confirmed by existing research (Palladino & Walker, 2007), the experimental paradigm I present in my work goes beyond Finlayson or Palladino and is related to the question, “How do we measure auditory cue comprehension?”

One approach for analyzing auditory cues is to use multiple choice tests (e.g., Finlayson & Mellish (2005)). Using this technique, we can determine whether a participant chose section of code A or B; however what regions of the code were critical for comprehension is unclear. Further, multiple choice studies scale poorly to lengthy computer code. In contrast, I have developed an evaluation metric called artifact encoding that overcomes these weaknesses.

The basic idea of the artifact encoding paradigm used in this dissertation is to give participants audio, and then to have them interpret the audio in relation to a run of computer code. The first step of this process, listening to audio and writing it down, serves several purposes. First, having a user
write down the audio itself eliminates the possibility that the audio was misheard or not heard at all. If the user did not write down a particular piece of audio, we immediately know, in the grading of the participant’s data, that that particular audio was missed. We can then choose whether to include the missed construct in the interpretation analysis.

One immediate question is, can non-sighted programmers participate in an Artifact Encoding study? The short answer is yes. While the current version of my empirical work uses sighted proxies, who can write answers on paper, Artifact Encoding studies can be ported to computer based tasks by making accessible web pages and re-piloting the time between specific auditory cues. Differences in the time needed between auditory cues may occur because non-sighted participants may require time to navigate a web page to give their answer. Having the tasks completed on the computer does have one additional advantage: \textit{complete automation of the study grading}.

Specifically, using a variant of the Needleman & Wunsch (1970) amino acid sequence comparison algorithm, we can derive a global alignment: a sophisticated string match between a participant’s answer in a study and the correct answer. While the details of how to accomplish this are complex, and will be discussed at length in Chapter 4, the end result is that determining how well a participant understood the trace of a computer program using auditory cues can be accomplished automatically.

Once users have written down the auditory cues related to one trace of a computer program, they are asked to interpret the meaning of the auditory cues and are given a few minutes to do so. During this time, participants literally translate the auditory cues into something very similar to computer code, essentially a trace. Once a series of tasks and traces have been completed, we code the data into a standard, highly reliable form (see Chapter 5), and then do statistical analysis on the results.
1.4.3 Experiment 1 - Auditory Cue Comprehension

The ground up approach promoted in this dissertation, as illustrated in Figure 1.3, uses both basic and applied research studies to try and improve the design of non-sighted programming environments. This first experimental study focuses on a human’s comprehension of auditory cues, the underlying assumption being that if an auditory cue cannot convey its meaning, then it is likely a poor choice in a non-sighted environment.

Specifically, this first experiment focuses on comprehension differences between cues that do or do not present relationships between program constructs—scoping. Thus, besides helping us to understand the comprehension of particular auditory cues, this experiment also helps to answer the question, “How do we measure auditory cue comprehension?”

In this first experiment, besides scoping cues, a series of additional auditory cues are output to a user. This allows the automatic grading paradigm to be able to detect other comprehension issues, including how well participants understood the behavior of `if` statements, loops at either various iterations or under various conditions.

Analyzing these raw comprehension issues is critical, even for real world programming tasks, as programmers who cannot comprehend the auditory cues cannot possibly program using those cues, at least not without significant training. For example, in Walker’s learning rate study, he found earcons took nearly 700% longer to learn than spearcons (Palladino & Walker, 2007) ². Artifact Encoding studies allow us to weed out auditory cues that are difficult to understand and also allow us to determine the circumstances under which those cues are difficult to understand.

The primary results from this study showed that (a) scoping cues of the form used in the experiment increase a human’s ability to understand scoping relationships by approximately 21% (b) scoping cues are helpful invariant of a programmer’s experience level, and (c) scoping cues do not negatively impact a human’s comprehension of a program as a whole (via metrics which will be

---

²While Walker did not compare spearcons to speech based stimuli in Palladino & Walker (2007), Walker et al. (2006) showed no differences between the two groups.
described in Chapters 4 and 5).

1.4.4 Program Execution and Debugging Environment

Development of a sonified debugger as an add-in or plug-in for existing environments (retrofitting) does garner accessibility, but building environments from the ground up (with empirical evidence as a guide), can dramatically increase usability. This environment addresses the question, “How can we develop an audio-based program execution environment that is more effective than the present state-of-the-art?”

Non-sighted programmers, due to problems like, “Where am I?” may greatly benefit from the ability to go backwards in the debugger. Other useful features include the ability to execute functions in their own program context, with no possibility of side effects, and the ability to cache information from any of the typical compiler phases, like static analysis. Ducassé (1999) and Lewis & Ducassé (2003) present an environment that caches every piece of state information over the lifetime of a program, essentially making it omniscient. The program execution environment created in this thesis is created similarly, and as such, it would be fair to call it a Sonified Omniscient Debugger (SOD).

1.4.5 Experiment 2 - Screenreader vs. Sonified Debugger

The first experiment was a formative study that analyzed comprehension issues using auditory cues (basic research). The second study was a summative study where users completed real world tasks (applied research). This experiment, via taking objective metrics of human performance with various versions of the Sonified Omniscient Debugger, also addressed the research question, “How can we develop an audio-based program execution environment that is more effective than the present state-of-the-art?”

In this experiment, programmers were given a computer program, told that the program has two bugs in it, and were then timed while they tried to (a) answer comprehension questions related to the state or behavior of the program and (b) find the two bugs. To make these tasks both realistic
and related to previous studies, this study compared two types of auditory cues: those generated from the screen reader JAWS using Microsoft Visual Studio.NET 2005’s debugging environment, and a set of cues generated from, and which expand upon, those used in experiment 1. These environments were additionally compared to a visual programming environment: the “gold standard” in how effective these environments can be.

Results from this second study showed that the Sonified Omniscient Debugger facilitated users ability to complete programming tasks. SOD was shown to significantly lower the time it takes for programmers to complete comprehension and debugging tasks when compared to a screen reader by approximately 65%. Surprisingly, the time to complete tasks in SOD was not statistically distinguishable from a visual environment, implying that it is possible to create non-sighted programming environments that approach visual performance.

1.5 Limitations

Studying non-sighted computer programmers is extremely difficult. The non-sighted programmer community is incredibly small compared to the sighted community. Furthermore, legal restrictions make obtaining contact information for non-sighted programmers challenging, and obtaining governmental statistics on the community is difficult. For these reasons, there are several limitations to the current research.

The most critical limitation of this work is that the empirical studies involved sighted proxies, who were either blindfolded or could not see the screen, instead of non-sighted programmers themselves. Ideally, we would want non-sighted programmers to critique designs, analyze tools, participate in surveys, and allow us to watch them. However, the reality was that we simply did not have access to non-sighted programmers. As has been done in previous work, we resorted to using sighted proxies (Finlayson & Mellish, 2005), and we cite the work of Leonard Poll as evidence that it might not matter. Specifically, in Poll’s initial feasibility test for his tool SoundTablet, where he tested 20 participants, 10 non-sighted and 10 blindfolded, he found no significant differences
between the two groups.

In this feasibility test, Poll tested both congenitally blind (blind from birth) and adventitiously blind (the person became blind later in life) in basic tasks where users manipulated a GUI application (Poll, 1996). Specifically, he had users, who were either non-sighted or blindfolded, a) localize auditory cues (he called them auditory objects) b) identify auditory cues, and c) directly manipulate objects represented via auditory cues. Direct manipulation, in this context, basically means drag and drop operations in a user interface, except that the user cannot see what they are dragging, nor where they drop what they are dragging.

While Poll’s work had nothing to do with computer programming, his research questions are quite related to the primary questions in my research. Localizing auditory cues is the same as asking, “Where am I?” and directly manipulating auditory cues is a navigation issue that I deal with implicitly in my debugger. There was no obvious equivalent to state in Poll’s tasks, but clearly, other researchers have identified the issues I discuss here as important to the non-sighted (Stevens, 1996; Smith et al., 2004).

So, while it would be ideal, for any researcher, to work with a large number of non-sighted computer programmers, it is generally not feasible. Further, evidence suggests that for certain types of tasks it might not matter anyway. In my case, I think this latter point is particularly important. Participants in my studies accomplish tasks that are related to how well an individual can understand speech based stimuli, which, in all likelihood, non-sighted users comprehend in the same as sighted users, barring additional medical conditions related to speech processing.

Another limitation of not working with non-sighted programmers directly is that it is difficult to verify the problems I am trying to solve are important for the non-sighted community. While existing research, like Poll, identifies similar problems, it’s possible that the problems addressed by this dissertation have been misidentified. However, one can mitigate the risk of choosing poor problems by interacting with the non-sighted community at large.

Specifically, I have been actively involved in a non-sighted mailing list for blind programmers,
available at www.blindprogramming.com. Typically, before conducting a study, I ask the community about various auditory cues, what their major problems are (a tremendous source of new study ideas), and general questions about how they program (e.g., JAWS scripting). Work that builds upon my own, if it eventually becomes feasible, should try to run formal empirical work testing non-sighted programmers. Such work can build upon the testing paradigms I am creating here.

1.6 Organization of the Thesis

Throughout the rest of this thesis, I will detail a design space of auditory cues for programming a computer (see Chapter 3), present an empirical paradigm for analyzing the comprehension of the cues used in SOD (see Chapter 4), present a formal study on a selection of behavioral runtime cues (see Chapter 5), present a detailed analysis of the Sonified Omniscient Debugger (see Chapter 6), test SOD in realistic comprehension and debugging tasks against a state-of-the-art screen reader (see Chapter 7), and finally sum up the contributions of this work (see Chapter 8). Before any of this, however, I will first discuss the existing literature related to creating auditory interfaces.
CHAPTER 2
LITERATURE REVIEW

In this chapter, I review the literature relevant to my primary goal: designing effective tools for non-sighted computer programmers. This chapter is broken into several primary areas, auditory display, aesthetics, technology for non-sighted users, comprehension, and general human issues. The literature most relevant to this dissertation is in the field of auditory display, a research discipline focused on presenting information via auditory cues. Throughout the rest of this section, I highlight these areas, discussing how they relate to the current work and pointing out where additional work is needed.

2.1 Auditory Display

The work I present in this dissertation falls squarely within a field known as auditory display. Auditory display has been used for a variety of purposes, including assisting programmers in debugging tasks (Vickers & Alty, 2003), helping non-sighted users navigate hierarchical data structures (Smith et al., 2004), and providing a richer and more informative desktop operating system interface (Gaver, 1989).

The field of auditory display is particularly important for the non-sighted. Research in this field covers the spectrum of using audio to display information: from relatively basic research on the perception of sound (e.g., auditory scene analysis (Bregman, 1994)), to work on very specialized applications, like understanding the structure of a graph with audio (Nees & Walker, 2007).

Regardless of the applications of auditory display, there are four primary categories of auditory cues: auditory icons, earcons, speech, and spearcons. These categories of auditory cues, and under what conditions they have been used, will be discussed next.
2.1.1 Categories of Auditory Cues

One important concept in auditory display was first defined by Gaver (1986) when working on the SonicFinder, a perceptual mapping. Perceptual mappings provide a conceptual framework for categorizing and understanding the pros and cons of various types of auditory cues. The most oft cited component of this work is Gaver’s categories of perceptual mappings: symbolic, metaphorical, and iconic (Gaver, 1989) or nomic (Gaver, 1986).

A symbolic mapping has meaning only by convention, and must be learned and remembered, like a language. A metaphorical mapping makes use of a non-literal relationship between an object and its representation. For example, the operation of copying a file from one part of a file system to another can be represented by the sound of water filling up a drinking glass.

An iconic, or nomic, representation looks or sounds like the thing it represents. For example, a nomic mapping of deleting a file, by throwing it in a virtual recycle bin, could be represented by the sound of throwing physical objects into a real recycle bin (Gaver, 1986). Determining an appropriate mapping for program constructs, however, takes considerably more imagination.

Gaver (1993) later worked on representing “dimensional” information in sounds, and proposed using sound synthesis techniques to parameterize the sounds. He presents this approach to give auditory icons, what this kind of auditory cue is called, more parameterization capabilities. For example, a file could sound large if it is large, or could sound small if it is small (Gaver, 1993). Auditory icons are not used in this dissertation, although they may be useful to indicate to a blind programmer certain kinds of error conditions.

Instead of sonification using auditory icons (symbolic mapping), earcons have been proposed as an alternative method for presenting information using audio (metaphorical mapping). Earcons are defined as “... nonverbal audio messages used in the user-computer interface to provide information to the user about some computer object, operation, or interaction” (Blattner et al., 1989). In this context, Blattner considers computer objects to be things like files or menus and operations...
to be things like editing or compiling. Editing a file would be an example of an interaction that could be sonified with an earcon. Most of the time, earcons are based around a musical structure.

The first empirical studies conducted on earcons were completed by Brewster et al. (1993). Later work found that participants’ ability to understand concurrent icons is somewhat limited (McGookin & Brewster, 2004, 2006), even when auditory scene analysis (Bregman, 1994) techniques are employed. Besides menus, earcons have been found to significantly increase the effectiveness of other GUI interfaces, like tool palettes, while not increasing how subjectively “annoying” the interface is (Brewster & Clarke, 2005). The annoyance factor, while it may appear trivial, is actually quite important for any individual who has to regularly make use of the technology.

While the use of non-speech audio is interesting, there is growing evidence that non-musical sounds are easier to learn (Palladino & Walker, 2007) and, in general, more effective than sonification or earcon based technologies (Walker et al., 2006). Specifically, while Walker’s latest work is predominately on spearcons (compressed speech), in a recent study, speech was also found to be more effective than earcons (Walker et al., 2006). Walker’s studies are, however, not directly related to programming, but instead focus on using menus in a menu system. In their auditory cue studies, Walker et al. measured learning rate (Palladino & Walker, 2007) and accuracy (Walker et al., 2006), and the time to complete various tasks (Walker et al., 2006).

Further, while the work of Walker and his colleagues experimentally found that spearcons are more effective and easier to learn than earcons or auditory icons (Gaver, 1986, 1993), they did not find evidence that they were more effective than speech. This might imply that speeding up the text to speech will not have a detrimental effect on the interpretation accuracy of certain tasks. Since Walker only tested his hypothesis for menu tasks, it is unclear whether this will hold for cognitively complex tasks like comprehending computer programs.

Which category of perceptual mapping does this final work on spearcons and speech fall into? The answer is not clear. On the one hand, speech based sounds use language—which was learned at some point (symbolic), but for a human adult, is it really fair to call a language one already
knows a symbolic mapping? Perhaps not—an already known language might be better classified as nomic, despite the fact that, for example, the sound of the word “dog” does not have a literal relationship to a physical dog.

On the other hand, perhaps Gaver’s categories of perceptual mappings should be entirely abandoned for more scientific and comprehensive categories (e.g., comprehension measures (see Chapter 4) or semantic priming (McNamara, 2005)). No matter the category of perceptual mapping speech or spearcons falls into, speech is overwhelmingly the auditory cue type of choice by the modern blind computer programmer—although researchers have tried many other cues, as demonstrated in the next section.

2.1.2 Auditory Display for Computer Programs

Sonnenwald et al. (1990) described an architecture for InfoSound, an auditory display tool for computer programs. Sonnenwald used several types of auditory cues, including speech, everyday sounds like a telephone ringing (auditory icons), and even musical sounds (earcons). These sounds corresponded to various events, where a telephone ringing could, for the most obvious presentation, represent someone calling on a phone. Music can, instead, represent events that are more difficult to define with a telephone ringing, or other every day sounds.

Brown & Hershberger (1991, 1992) discussed algorithm animation using both color and sound, of which the sound portion is most relevant here. The sounds used were predominately music based, with some occasional Gaver-like natural sounds intermixed with the auditory cues (also referred to as auralizations). For example, one auditory cue created was for a selection sort algorithm. In this case, a muted clarinet was used to represent a comparison event, a gong to represent exchanging data in the array, and a xylophone sound to represent what they call the “BestSoFar” event, which basically means when a piece of data in the array is found that best fits the current iteration of the selection sort, like the lowest numeric value. While techniques for representing individual algorithms in sound are interesting, later work, by Finlayson, Vickers, and myself, have
tried to make more general purpose, algorithm independent, audio replacements for traditional visual stimuli.

Boardman et al. (1995) created the language Listen, now reimplemented in Java and called JListen. The original motivation for Listen was to create a tool for describing how computer source code can be sonified. This work included the creation of the LSL, Listen specification language, that allowed auditory cues to be put into other computer programs. This specification tells the Listen system how to interpret computer code it receives, and what to sonify in that code. So, in Listen, the focus was on inputting code and outputting code with auditory cues embedded. In my work, however, I focus on measuring the human effectiveness of auditory cues.

Vickers & Alty (2002b, 2003) used musical structures to design their auditory cues. With the program, CAITLIN, Vickers describes his method of auditory display for the Pascal programming language. Vickers ran empirical studies using his auditory cues (Vickers & Alty, 2005). The main effects of his experiment, that participants could find more bugs with musical cues was non-significant, although Vickers claims there may be a correlation between the benefits of musical cues and the cyclomatic complexity (Elshoff & Marcotty, 1978) of the source code.

Begel & Graham (2004, 2005, 2006) created Spoken Java, a tool designed to allow users to speak computer code instead of typing it. This work can be fairly characterized as the reverse of my own. In Begel’s work, speech is input into the computer to help the user program, while in my own work, speech is output from the computer to indicate information about a program. Begel’s work is reminiscent of Stevens (1996) work on Mathtalk, where the way users read speech was analyzed to determine appropriate prosodic cues for reading mathematical formulas to blind users.

Finlayson & Mellish (2005) showed that speech-based audio was more effective for computer programming based tasks than Vickers style earcons. They used simple earcons in a multiple choice type study to measure whether participants could better distinguish sections of computer code with either technology. Speech was found to be more effective, which is not terribly surprising considering humans have a lifetime of interpreting speech, but little to no experience interpreting
music. The measurement paradigm used by Finlayson & Mellish (2005) was, however, rather crude. The pros and cons of their method will be contrasted with my method (artifact encoding) in Chapter 4.

Berman & Gallagher (2006) presented work on the sonification of program slices. Berman’s work has two unique features. First, Berman sonified program slices, which by their nature are static, unlike the dynamic comprehension issues presented here. Second, Berman used non-tonal sound structures as auditory cues. Most musical auditory cues in the literature, in fact nearly all based on earcons, use extremely basic tonal music as the basis for their work. Berman draws upon a richer musical structure, including techniques like granular synthesis (Dodge & Jerse, 1997), which are commonly described as “clouds” of sound.

2.1.3 Discussion

So, what do we know about representing computer programs using audio? Unfortunately, since the literature still predominately uses anecdotal evidence, I would argue that we do not know much. There does, however, seem to be growing evidence that speech based sounds are easier to learn than musical sounds, and researchers are increasingly testing similar hypotheses empirically.

I would ask, however, which speech sounds are more effective? In Chapter 7, I show two programming environments that vary only by what speech is output to the user, yet there is a significant difference between the environments. In short, good speech is effective, yet the literature provides little guidance on how to define what good is, and existing methods for measuring the quality of auditory cues sometimes have significant drawbacks (See Chapters 4 and 5). Artifact encoding, the experimental technique used in this dissertation to analyze the comprehension of an auditory cue, fills this gap by giving us a method for analyzing the goodness of an auditory cue. Artifact encoding tells us how well someone understands an auditory cue, whether it is an earcon, spearcon, auditory icon, speech, or something else entirely.

Auditory display has encompassed a wide array of interfaces and problem spaces. Tasks vary,
from learning to program a computer using audio, to navigating in a virtual environment, to glancing over the structure of a graph. There continues to be a debate as to the best way to represent sounds: either with speech, music, or sonification proper.

Results discussing whether earcons, spearcons, speech, or auditory icons are more effective for specific tasks is important. Audio based technology holds inherent, and significant, challenges for the user and not all tasks can be considered equal *a priori*. Designers of any auditory interface must carefully minimize the demands on working memory (Baddeley, 1992) by choosing auditory cues that continuously remind the user what is happening in a computer program. Different strategies appear to be more, or less, effective depending upon the specific needs of the interface. As such, the debate as to whether speech or non-speech audio is more effective will likely continue for many years to come.

2.2 Aesthetics

In order for auditory display techniques to enter into the mainstream of the common interface toolkit of developers, auditory cues must be both effective and aesthetically pleasing. Current auditory display technologies are often based on primitive musical structures (Brewster et al., 1993) or simple metaphors (Rigas & Alty, 2005; Alty et al., 1997; Alty & Rigas, 2005) that, while sometimes effective, are not particularly interesting to listen to. Even sonifications based on musical structure (Vickers, 1999), are more akin to a music theory exercise than a Beethoven string quartet. The priority scientists have taken—effectiveness before pleasantry—makes a great deal of sense, but contemporary researchers in auditory display are beginning to consider both priorities. The basic question researchers with the aesthetic slant ask is, “How can we make our interface both effective and pleasant?”

In Vickers (2004) discussion of aesthetic computing, he pointed out the need for a better user experience in sonified interfaces. In Vickers later work on Auditory Display systems that consider aesthetics (Vickers & Hogg, 2006), he discussed a continuum from *Ars Informatica* to *Ars Musica*,

25
where the former considers sonifications that are designed to give information, and the latter end is a set of sounds with no information being presented. He argued that, “the interesting area lies in the middle where sonifications have been deliberately designed with artistic sensitivities (Vickers & Hogg, 2006, pg. 213).” This line of argument appears to imply a trade-off between a sonification being both informative and interesting to listen to.

Leplâtre & McGregor (2004) presented a case study designed around an email notification system. Several different types of auditory cues were categorized and an experiment was run, the goal of which was to discover aesthetic differences between a variety of sounds. The results of this work suggested that the type of task completed by a user has an affective effect.

Another equally difficult problem to consider is how to describe sounds of various types. Different types of users can describe sounds in a variety of ways, although the number of parameters, length, volume, timbre, or others do not always have an intuitive verbal explanation. Work on soundscapes, perceived auditory environments (McGregor et al., 2006), has been undertaken to try to determine appropriate verbal descriptions.

The work of Lazar et al. (2006a) investigated the relation between a user’s mood and frustration, specifically for non-sighted users. In this work, 100 non-sighted users logged their overall mood while using a computer to browse the web with a screen reader. In their hierarchical regression analysis of frustration they found that six primary factors affected a non-sighted user’s change of mood when using the computer: overall frustration, relation between frustration and completion of their intended work, prior experience with the Internet, overall anxiety, having sufficient hardware or software for their needs, and whether they continued to ponder problems they were having after computer use. Unlike sighted computer users (Lazar et al., 2006b), non-sighted users’ mood is not affected by the amount of time lost when completing a task (Lazar et al., 2006a). Perhaps this is due to the already laborious and tedious nature of using a screen reader. In other words, perhaps non-sighted users are in the unfortunate predicament of being accustomed to poor and ineffective interfaces.
Lee et al. (2000) conducted a study on user preferences for sounds used in the design of a portable microwave oven. The sound design in a microwave oven is a remarkably good example of the care required for a good sound based interface, especially for non-sighted users. The user must tell the machine how long to cook an item and, if non-sighted, must receive auditory feedback for that length. The user must also be told several events, like completion of cooking or turning the unit off. While Lee et al.’s study was somewhat informal, they carefully considered the affective nature of the interface, in addition to its effectiveness.

In sum, a growing amount of work is considering the affective relationship between computers and their users. Users with disabilities may be affected differently from those without (Lazar et al., 2006a). For sound based user interfaces, taking subjective measures in addition to measures of overall effectiveness, (see e.g., Brewster & Clarke, 2005), may be important, especially after we know more about what makes sound based interfaces effective or ineffective. Auditory displays might be on a spectrum from Ars Informatica (informative but boring), to Ars Musica (pleasant but useless), although the relationship between whether making a sonified interface more affective reduces, or increases, effectiveness for various tasks is yet unclear.

2.3 Technology for Non-sighted Computer Users

Unlike sighted users, non-sighted programmers must use a screen reader in order to program. In this section, we explore the contemporary literature on non-sighted users in general. While little work exists on non-sighted programmers, studying screen readers, technology for the web, or other interfaces for the non-sighted can give insight into how to design my programming interfaces for the non-sighted.

Designers have created technologies to help non-sighted users accomplish a variety of tasks. Tools have been created to help users navigate (Lutz, 2006), navigate web pages through audio enriched links (Parente, 2004), read tables (Yesilada et al., 2004), identify geographical information using non-speech audio (Zhao, 2005), adapt GUI applications for audio (Parente, 2006), browse
molecular structures (Brown et al., 2004), and to enhance short term memory in non-sighted children (Sánchez & Flores, 2004). Even with extensive research in a variety of areas that would be potentially useful to non-sighted users, screen readers are still the predominant technology in use. Integrating them into the educational curriculum, even on a limited scale, is gaining some attention (Freire et al., 2007).

Kildal & Brewster (2007b,a) used speech based audio for reading tables to non-sighted computer users. Although full experimental studies have not yet been conducted on their tool, TableVis, they found anecdotally that reading long lists of numbers taxes working memory. For reading long lists of numbers, some kind of truncation needs to be employed in order to give an overview, or glance (Stevens, 1996), of a document. The exact form this truncation should take is relatively unexplored, but the issue also comes up in computer programming, where the tool must read the value of variables at runtime. This situation is especially common in programming when using floating point values, where the user, as Brewster suggests, needs an overview and not the specifics.

Stevens (1996) presented a methodology for using text to speech for reading algebra to non-sighted users. In Stevens’ work, the core idea was to determine appropriate word choices and to use prosody to determine how algebra should be spoken. In Stevens’ work, users were asked to read algebra, the prosody of the voice was measured, and the results from the analysis were incorporated into the design of the algebra reader. The current thesis focuses predominately on word choice; however, adding prosodic cues could be a beneficial addition to the current work.

Work by Parente (2004) is similar to Stevens’ work on audio glances. In his work on audio enriched links, Parente created a tool that can summarize, and preview, a link in a web page before a non-sighted user clicks on the link. Likewise, the auditory cues presented in this thesis attempt to summarize information regarding a computer program in as succinct a way as possible.

Raman (1996) presented Emacspeak, a tool that was intended to replace the modern screen

---

1 Prosody is the use of voice inflection, dramatic pauses, and word emphasis.
reader by integrating speech based audio into applications, specifically for the non-sighted. Emacspeak is relevant to the current work, as the sonified debugger created in this thesis directly adds spoken audio into a modern program execution environment, as opposed to using an external application, like a screen reader, which cannot gain context from the program directly. Raman did not, however, use empirical studies to guide the design of the speech based audio system. Further, emacspeak was a general purpose application, and did not focus specifically on how the tool can be used for programming.

Stallmann et al. (2007)’s Proofchecker system is a finite automata builder and checker used in classes at North Carolina State University. In these classes, students are given assignments to create certain types of automata and these automata are checked by the computer for correctness. While this program was originally designed for sighted students, an accessible version of ProofChecker was recently created. While Stallmann et al. (2007) do not study in any detail the effectiveness of the accessible version of ProofChecker, which essentially uses Java’s accessibility API and the screen reader JAWS, the important point in this work is that a growing number of accessible applications are being created for all levels of the computer science curriculum, from development environments like my own, to finite automata theory in Stallman.

King et al. (2004) and King (2006) created tools that read complex diagrams to the blind, including those written in the Unified Modeling Language (UML). King (2006) studied different methods for interacting with the auditory cues that represent the UML diagrams and found that the use of spatial cues are less useful to the blind than the use of auditory cues that represent logical relationships. Using terms from this thesis, King’s auditory cues would be considered a form of design cues.

The modern approach to providing non-sighted accessibility is to retrofit sighted applications with a screen reader. These retrofits are detrimental to non-sighted usability—as they often use auditory cues that convey little to no meaning to a non-sighted user. In stark contrast, I argue that non-sighted interfaces should be built from the ground up—via empirical studies that guide the
effectiveness of such interfaces. As will be explained in Chapter 4, my approach to building non-sighted programming interfaces from the ground up is artifact encoding, a paradigm for measuring the comprehension of auditory cues.

2.4 Program Comprehension

Comprehension of computer programs is a well studied task, including static (Pennington, 1987b,a), dynamic (Mayrhauser & Vans, 1996), and large scale (Mayrhauser & Vans, 1994) analysis of computer code. Often, comprehension is analyzed using protocol analysis (Letovsky, 1986). Protocol analysis, or similar techniques, are notoriously time consuming. As such, empirical work is typically done on only a few participants (see e.g., Kelly & Buckley, 2006).

Mayrhauser & Vans (1994) present a model of program comprehension. According to this model, there are three mental modes programmers engage in when programming: top-down structures, program model structures, and situation model structures. Top-down structures are concerned with large scale abstractions–for example, a component in an operating system that handles threading (Mayrhauser & Vans, 1997). Little work exists on directly sonifying components at the top-down level of abstraction, but see King et al. (2004); King (2006), although some work exists on the sonification of graphs or similar structures (Nees & Walker, 2007; Brown et al., 2006), which could be applicable.

In contrast, the program model originally discussed by Pennington (1987b,a), is concerned with mid-level abstractions (e.g., control flow). The auditory display techniques in this thesis are predominately concerned with program model abstractions based on Pennington’s finding that when code is new to the programmer, these abstractions are built first. Since Pennington’s work suggests program models are important to developers who work with unknown code, my empirical tests have focused on these program model abstractions first as well.

The last part of the Mayrhauser & Vans (1997) model, the situation model, is related to data
flow or functional abstraction concerns. While these concerns are important for program comprehension as well, mapping data flow to auditory display techniques for programming is rare. To my knowledge, the only empirical study using auditory display techniques for data flow, at runtime, in programming was in my own work (Stefik et al., 2007). Little was learned from that work, in relation to data flow, other than learning that participants’ had difficulty following even one variable, under strict conditions, using auditory display. For static representations of source code, Berman’s work on sonifying program slices is about as close to the situation model as the current literature comes (Berman & Gallagher, 2006). Substantially more work is needed in the auditory display of situation model concerns.

Aural representations of source code (Vickers & Alty, 2002b,a, 2003, 2005; Lapidot & Haz- zan, 2005; Stefik et al., 2006b,a, 2007), may invoke different comprehension strategies. Parallel computer programs may require different strategies still (Francioni et al., 1991b,a; Francioni & Jackson, 1993). The aural medium is, by its nature, serial relative to time. This serial nature of audio is both a great advantage and disadvantage, depending upon what type of data the participant is trying to understand. For example, in Smith et al. (2004)’s hierarchical structure analysis tool (Smith et al., 2000; Francioni & Smith, 2002), or Brewster’s studies on earcons for hierarchical data (Brewster et al., 1996), the audio must give sufficient clues for users to locate their position in a hierarchical structure. With any hierarchical structure tool for the non-sighted, the interface must answer the question, “Where am I?”

Determining how humans comprehend computer programs involves a series of complex questions. How we comprehend programs is dependent upon the size of the code base (Mayrhoaser & Vans, 1994, 1996, 1997) and our experience level (Pennington, 1987b). Understanding code using visual structures is different from understanding it using audio only structures (e.g., visual interfaces allow users to see multiple items nearly in parallel, while auditory interfaces are predominantly serial). A great deal more work is needed in order to answer the basic questions of how programmers interpret code, how they write and maintain code, and how having disabilities, like
lack of sight, alters strategies for comprehending computer programs.

2.5 Human Issues

In this section, I review some of the major work in how humans process information, aurally and visually, and limitations on mental storage. While the work presented here does not directly relate to program comprehension related tasks, human limitations can and should inform the design of any computer-based tool. I focus here predominately on psychological research that can inform the design of the audio based program comprehension tools presented in this thesis.

Human memory not only has an influence on the comprehension of computer programs, but also has an effect on the experimental tasks used in this dissertation. Baddeley (1992), gives a short, but thorough, review of working memory models. The core principle in Baddeley’s work is that there are three primary components in working memory: the phonological loop, the visual sketchpad, and the central executive. This work was later expanded to include a fourth component, the episodic buffer (Baddeley, 2000). Figure 2.1 sums up the components; in the next few paragraphs, each of these components is described in more detail.

The phonological loop is a component of human memory related to speech and speech processing. Baddeley divides the phonological loop into two primary components, the phonological store and the articulatory store. The phonological store can hold speech information for only a few seconds, although this can be extended if the articulatory control process (continual mental rehearsal), mandates it.

Further, Baddeley recognizes four primary effects that occur in the phonological loop. The first effect is called the acoustic similarity effect. This effect states that if words sound similar they are harder to remember. The second effect is the irrelevant speech effect, which states that nonsense words can affect the phonological store, lowering recall even though they have no semantic meaning. The word-length effect states that, “Subjects can generally remember about as many

\footnote{Reprinted from Journal of Communication Disorders, Volume 36, Alan Baddeley, Working memory and language: an overview, Pages 189–208, Copyright 2002, with permission from Elsevier.}
words as they can say in 2 seconds. (Baddeley, 1992, pg. 558)” With such limited memory, audio based programming tools must limit individual phrases to be less than 2 seconds long. Articulatory suppression, the last of the effects discussed by Baddeley regarding the phonological loop, is an attempt to disrupt mental rehearsal (called subvocal rehearsal). The idea is to ask participants to mentally rehearse a given word repeatedly. It is unclear whether this effect has any relation to audio based integrated development environments, but it definitely is the case that different types of auditory disruptions can cause an impact on performance in certain tasks (Tsujimura & Yamada, 2007).

The last three components discussed by Baddeley are the visual sketchpad, the central executive, and the episodic buffer. The visual sketchpad integrates “spatial, visual, and possibly kinesthetic (Baddeley, 2002, pg. 200),” information for storage and processing. The central executive is a main controller and does not store information directly (Baddeley, 2000). Unlike the central executive, which is considered to be a controller, the episodic buffer is an integrator. The episodic
buffer takes information from other components in the mind and is controlled by the central executive (Baddeley, 2000).

Besides issues of human memory, an effect known as *semantic priming* may play a role in how humans process speech based auditory cues. McNamara (2005) discusses the recent literature of semantic priming—a theory which, at its core, implies that if a user is given a particular stimuli (e.g., a word, perhaps: robin, bird, or cat), that word can prime (either facilitate or inhibit) a participant’s ability to complete tasks on a second word (e.g., lexical decision tasks on the word “dog”). This effect does not always occur and is strongest when words are related (e.g., cat and dog), implying a mental form of *spreading activation* in the mind. In computer programming, the syntax of a programming language naturally acts as a set of primes; in non-sighted programming, the auditory cues act similarly.

Other issues related to how humans process audio or speech may have an effect on human performance when using audio based programming environments. For example, auditory temporal masking (Zhang & Formby, 2007), the theory that a sound stimulus can influence the audibility of another stimulus either before or after the original, may also have an effect on how non-sighted users process information in a sound based programming environment. Since, in programming, a number of superfluous characters are often embedded into an auditory stream (e.g., parenthesis, brackets, braces), the combination of auditory temporal masking and the irrelevant speech effect could significantly hinder human processing of auditory information from a programming environment.

Whitney (1998) gives a thorough overview of how humans process speech and language. The details of speech processing are incredibly detailed and complex, with research issues that range from the minutia of speech, like how humans process individual phonemes of speech, to how humans understand large scale structures, as is studied in discourse processing. Since the program execution and debugging environment presented in this thesis is entirely speech based, carefully constructing the speech interface implies taking human processing concerns into account.
2.6 Conclusion

In this chapter, I have given an overview of the literature on auditory display, including aesthetic issues, general technology for non-sighted users, the comprehension of computer programs, and work related to human memory and speech processing. While this literature explores using sound for various applications, I would argue that one of the largest, if not *the* largest, need for the auditory display community is in creating new paradigms, measurement techniques, and experimental studies, regarding how humans actually use these auditory display interfaces. Throughout the rest of this dissertation, I propose and develop techniques as an initial step toward filling this gap. In the next chapter, I explore and classify various types of auditory cues for use in computer programming.
CHAPTER 3
THE DESIGN SPACE OF AUDITORY CUES

Computer programmers have to complete tasks at various levels of abstraction (Pennington, 1987a). Different tasks require different information. When designing a program, for example, programmers may need to know how classes relate in an inheritance hierarchy, or what kind of design pattern (Gamma et al., 1995) to apply. When editing, or executing, however, programmers likely have different information requirements. In this chapter, I present a novel design space for programming environments for the non-sighted. Figure 3.1 presents an overview of the design space, which consists of three main types of auditory cues: Design cues, Editing cues, and Execution cues. Since empirically testing all of these cues is beyond the scope of a single thesis, I have chosen to focus predominantly on a subset of execution cues, as will be discussed in Chapter 5.

3.1 Design Cues

Design cues indicate the architecture of a software system. This type of cue presents high level information, telling the programmer the abstract structure of the code, but ignores low level concerns like syntax. Users should receive design cues when they want to know information about the components in the system and how those components interact.

Probably one of the best examples of design cues in the literature is in the TeDub system, which is designed to present technical drawings to the non-sighted (Petrie et al., 2002). This work was expanded specifically for UML by King et al. (2004), who suggest that for programming applications, “Working item by item through the information content in an unstructured linear fashion with a screen reader will not be practical (King et al., 2004, pg. 524).”

King et al. (2004) provides two methods for navigating a UML graph while hearing design cues. The first method of navigation is a text based layout of the UML graph, where the user navigates via a tree based hierarchy built from a particular UML graph. Nodes in the tree that have
Figure 3.1: A summary of the types of auditory cues described in this thesis.

children play a “context” sound when navigated. A combination of earcons, auditory icons, and spatial audio were used as the context sounds (King, 2006).

This second navigation technique used a joystick to spatially navigate the UML. For example, users move the joystick in a given direction and the graph is polled to determine if another node is in that direction. If there is a UML node in that direction the program will navigate to that node. The problem with this approach is when multiple nodes have approximately the same direction, as the program has to determine which node the user meant. This problem may make navigating to some nodes more difficult than others, depending on the specific UML graph and its corresponding structure.

The fundamental lesson learned in King et. al’s work is that spatial information is not particularly useful for the non-sighted when interacting with auditory cues for representing UML. Instead, those creating design cues should focus on representing, and user interaction with, logical relationships, not spatial ones. Besides this issue, the ability to go backwards was found by King
to be well received in user evaluation.

3.2 Editing Cues

While design cues represent high level abstractions of source code editing cues represent low-level information. Programmers receive editing cues while modifying or writing source code.

Figure 3.2 defines two types of editing cues: language cues and localization cues. Language cues indicate the syntax and semantics of the computer program in question, while localization cues indicate the context under which the programmer is working. Each of these cue types will now be discussed in more detail.

3.2.1 Language Cues

I see two primary types of language cues that can be used when reading a line of text in a source code editor: syntax and semantics cues. Screen readers typically output syntax cues. Consider the statement \( a = * b * c; \) in the programming language C++. A syntax cue generated from this code would read the text verbatim, while a semantics cue would attempt to reveal the meaning of the statement.

There are several pertinent questions related to this syntax cue. First, should the character "*" be called an asterisk, as in “a equals asterisk b asterisk c semicolon” or should it be called star, as in “a equals star b star c semicolon?” The screen reader JAWS, given \( a = *b *c; \) reads the "*" character as “star.”

Word choices like this might seem trivial, but the word star is significantly shorter than the word asterisk, and as such, by default should be preferred. Recall that Baddeley suggests that approximately only 2 seconds of speech can be retained in the phonological store (Baddeley, 1992). Minimizing the amount of information placed in the audio stream is critical.

Word choice issues aside, suppose a syntax cue is used for a similar statement, \( a = *B *c; \) Since capitalization is important in most modern programming languages, the fact that the letter B is a capital needs to be indicated to the user. Modern screen readers accomplish this brilliantly
Figure 3.2: The top part of this figure shows modern visual tools and techniques for programming, from left to right UML, the Netbeans Editor, and the Netbeans debugger. The bottom represents the corresponding design space of auditory cues for the non-sighted programmer, which uses audio to replace visual stimuli.
through their use of prosodic cues.

The JAWS designers could have decided to read the text, $a = \ast B \ast c;$ as, “a equals star capital b star c;” but this might imply to the user that the syntax includes the word “capital.” Instead, JAWS says “a equals asterisk b asterisk c semicolon,” but when the letter “B” is stated, JAWS makes it sound as if the character “B” is being yelled to the user. While it is difficult to describe this distinction in written text, in actual audio, the prosodic cue is obvious.

On the other hand, users may not actually be interested in hearing the raw syntax of a line of code in every possible situation. It might be the case that a semantics cue, which ignores the raw text and, instead, indicates the meaning of a particular statement, is more helpful to the user. Suppose, again, that the computer is reading $a = \ast b \ast c;$ but this time using a semantic cue. This statement could, in order to indicate its meaning, be read as either, “a assigned deref b times c,” “a is set to dereferenced pointer b multiplied by c,” or maybe even “set integer a to dereferenced pointer integer b times integer c.” Determining, however, which part of the statement’s meaning should be preserved, and how much detail should be given about each component of the statement, is not always obvious.

For example, whether “deref” or “dereferenced pointer” should be used is not clear. Professional non-sighted programmers might not be bothered by the word deref, but dereferenced pointer might be more clear for beginners. On the other hand, “deref” could be just as confusing to professionals. The same goes for deciding whether to use the word, “integer” or “int.” Professional non-sighted programmers might benefit from the shorter words, while novices might benefit from the longer. Which version is actually more beneficial for either user group is an empirical question, and for a user group that must use auditory cues, the non-sighted, these decisions are important.

3.2.2 Localization Cues

In addition to syntax and semantics cues, I also define localization cues. A localization cue gives information related to the context the cursor is in. Sighted programmers can easily glance up
or down from this location to determine their context, whereas non-sighted programmers need alternative techniques. Thus, localization cues answer the question, “Where am I?” while the user is editing.

Localization cues are broken into three types: *code above*, *code below*, and *scoping* (see Figure 3.2). Code above, or code below, enable *glancing* (Stevens, 1996), up or down from the cursor. Scoping cues, in contrast, indicate scoping relationships between pieces of syntax. For example, if the user hears a “}” which corresponds to the closing brace of an if statement, a scoping cue might say something akin to, “right brace end if at line 10.”

3.3 Execution Cues

Unlike design cues, which present high level summaries to the non-sighted programmer, or editing cues, which present low level textual or semantic information, execution cues, as shown in Figure 3.3 represent information about a running computer program, including both state and behavior. For this reason, execution cues do not try to aurally represent issues related to syntax errors. I classify execution cues into four primary types: *querying*, *temporal localization*, *navigation*, and *runtime cues*.

3.3.1 Querying Cues

When sighted programmers want to determine the state of a computer program, they most often consult a *watch* window. Watch windows visually display the current state of data in a computer program. Users can easily glance at this window without navigating to it, allowing them to visually determine “What’s up?” Non-sighted programmers must, however, explicitly (a) navigate to this window, (b) navigate to the variable in question, and finally, once they have determined whatever state they are interested in, (c) navigate back to their original location in a program.

Since they can see both the watch window and the code at the same time, sighted programmers can understand the context under which a variable occurs and its value. In contrast, non-sighted
Figure 3.3: A more detailed view of execution related auditory display information.
programmers must aurally navigate the source code, or memorize the code, to determine the context of any variable. Thus, by querying execution cues, non-sighted programmers should be able to gain as much information about a variable or its context with minimal effort and without undue burden on working memory (Baddeley, 1992).

3.3.2 Temporal Localization Cues

In addition to querying, non-sighted programmers also need some way to determine where they are located in a running program, temporal localization cues. Localization cues indicate the context under which the cursor is located, while temporal localization cues indicate information related to an executing program. Examples include recursive depth, call stack information, or what line of static text will be executed next in the debugger.

As an example of temporal localization cues, consider Figure 3.4, a typical recursive algorithm for computing a factorial. If the parameter passed in is 1 or less, the algorithm has reached its base case and halts, otherwise it recursively calls itself, continually subtracting one from the parameter on each call. A syntax cue for this algorithm might indicate that the cursor is at the line `if (1 >= n) {`. However, this tells the user nothing about the runtime nature of the code. A localization cue might tell the user that the cursor is within the method `factorial`.

In Contrast, temporal localization cues might tell the user how many times `factorial` has been called, how many activation records of `factorial` are currently on the stack, or it might allow the user to browse those activation records. In short, temporal localization cues give the programmer the context under which the program is executing, essentially, “Where am I?” at runtime.
3.3.3 Navigation Cues

Navigation cues are closely linked with runtime cues; both indicate the state or behavior of a program. Navigation cues tell the user what the program is doing, or what it did, as they navigate the runtime execution of a program. As an example, in Visual Studio 2005 using JAWS, when the user presses the F11 key, which tells Visual Studio to step over in the debugger, JAWS says, “F11.” While this cue tells the user what key was pressed, it tells the non-sighted programmer nothing about what effect F11 had on the state or behavior of the program (e.g., in artifact encoding, the comprehension score of this cue would be zero).

Ideally, Navigation cues would do more then tell the user what key was pressed, since this information is only marginally useful. More than likely, the user should at least be told the operation the computer is currently doing (e.g., stepping over, back, into, out). When keys are pressed, it might be even more helpful to use runtime cues, as demonstrated in Chapter 5.

3.3.4 Runtime Cues

Runtime cues indicate the state or behavior of a program as it executes. These cues are broken into two categories: value cues and behavior cues. Value cues represent the value of variables when actions take place. For example, consider Figure 3.5. Value cues represent the value of variables in this statement as it is executed. A cue might be, “a, 1, plus b, 3, is equal to c, 2, times d, 2.” Alternatively, value cues might tell information about the type of objects in use, the values of variables in objects, or the value of different elements in an expression.

Behavior cues, in contrast, tell only a statement’s behavior, not the reason why that behavior occurred. For example, for the statement \((a + b) == (c \times d)\), if \(a + b\) really did equal \(c \times d\) at runtime, then the behavioral cue would be, “if true.” Loops and other statements are accorded similar cues. With behavioral cues, loops might be given information related specifically to whether the conditional in the loop was true or false, or how many times the loop has executed.  

---

1If a loop had the auditory cue, “While true,” this would be a purely behavioral cue, but if the loop had a cue like “While iteration 1” this might be fairly considered a hybrid between a behavioral cue and a temporal localization cue.
if (a + b == c * d) {
    return 1;
}

Figure 3.5: A simple if statement which, at runtime, can have a number of auditory cues associated with it.

3.4 Conclusion

Auditory cues come in a variety of styles. Auditory displays can represent high level information using design cues (see e.g., King et al. (2004)), low level syntactic information (e.g., the syntactic cues offered by modern screen readers), or information related to the runtime state or behavior of a program, (e.g., the cues output by the Sonified Omniscient Debugger). Many of these cues can take a wide variety of forms, including different word choice, prosodic components, or even non-speech audio.

It is important to recognize that no single type of cue is necessarily better in all contexts. Syntactic cues have limited utility when the user is trying to determine information about an executing loop, while information about an executing loop will not help users determine if they forgot a semicolon. Context is important. In the next chapter, I begin exploring empirical methods for analyzing the design space presented here, including a paradigm for analyzing human comprehension issues with auditory cues: artifact encoding.
CHAPTER 4

EXPERIMENTAL PARADIGM: ARTIFACT ENCODING

The most fundamental facet of an auditory environment built from the ground up is whether the user understands the meaning of the audio. Poor comprehension of auditory cues trumps any other design consideration. Users can neither interact with, nor modify, components of a system that they do not understand. But how do we measure the comprehension of an auditory cue? My approach is called artifact encoding.

Artifact encoding is an experimental paradigm for analyzing how humans comprehend computer programs using audio based stimuli. Artifact encoding uses free-form writing and coding to try and estimate an individual’s understanding of audio stimuli. The advantage to this system is that it generates a great deal of data and, once the data is compiled, it can be automatically analyzed by a computer. Previous approaches tend to not give enough information about a user’s comprehension (e.g., multiple choice studies), or are too complex as a paradigm to specifically study comprehension (e.g., usability studies).

Throughout this chapter I will, first, discuss previous work in experimental paradigms for analyzing sound based computer interfaces. Then I will move on to artifact encoding, including the experimental procedure, how data is coded, and how it is graded.

4.1 Previous Empirical Approaches

While artifact encoding uses free form writing tasks to understand a participant’s interpretation of auditory cues, other researchers have adopted different approaches. In this section, I discuss three previously-used methods for determining the effectiveness of auditory cues: multiple choice studies, usability studies, and field studies using software engineering metrics.
if( a == b ) {
}

Figure 4.1: An if statement

<table>
<thead>
<tr>
<th>Choice</th>
<th>Program Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>if statement</td>
</tr>
<tr>
<td>B</td>
<td>while loop</td>
</tr>
<tr>
<td>C</td>
<td>assignment statement</td>
</tr>
<tr>
<td>D</td>
<td>expression</td>
</tr>
</tbody>
</table>

Table 4.1: A table with multiple choice options that may identify the code represented in Figure 4.1.

4.1.1 Multiple Choice

One easy way to determine whether a particular auditory cue, or a set of auditory cues, is effective is to use multiple choice studies. For example, Finlayson & Mellish (2005) had participants dis-ambiguate sections of computer code that were represented through audio. Participants listened to an auditory cue and chose a letter (e.g., A, B, C, or D) that they felt most closely represented the audio that they had heard.

The largest benefit of study designs based on multiple choice paradigms is that, once data is collected, the data is extremely easy to grade and statistically analyze. The biggest drawback, however, is that this study technique is *data poor*. In this context, data poor means that as the number of auditory cues given to participants increases for a particular question, the amount of information researchers can obtain regarding the comprehension of those cues decreases. To clarify this point, consider the computer code presented in Figure 4.1. A multiple choice study could be designed, for example, in which speech/musical auditory cues were presented to a participant, after which they could choose amongst the program constructs listed in Table 4.1.

Now suppose participants had to choose amongst four multiple choice options that indicated the code in Figure 4.2. Choosing A, B, C, or D might give some idea of how the participants understood the cues, but the longer the cues and the code, the less the participant’s marking tells the researcher. Thus, while multiple choice studies have their uses, especially for quickly testing
int main() {
    int a = 0;
    int b = 1;
    int c = 2;
    int d = 3;
    int i = 0;
    if(a == a) {
        if(a != a) {
            }
        else {
        }
    }
    else {
    }

    if (a == a) {
        if(a != a) {
        }
    }
    else {
        i = 0;
        while(i < c) {
            if(a > b) {
            }
        }
        else {
            i = i + 1;
        }
    }

    if (a == a) {
        if(a == a) {
            i = 0;
            while(i < b) {
                i = i + 1;
            }
        }
    }
    return 0;
}

Figure 4.2: If a participant wrote A, B, C, or D when choosing amongst 3 similar examples, there is no way to know what part of an auditory cue actually aided comprehension.
small code examples, scalability is a significant issue—one that, as we shall see, is mitigated by using artifact encoding.

4.1.2 Usability Study

Use in Measuring Comprehension

Another approach to analyzing auditory display based interfaces is to conduct usability studies. While many approaches to usability testing exist, the basic idea is to analyze participants’ performance via whatever metric is relevant for that type of tool. For example, one approach to studying auditory interfaces could be to use wizard-of-oz, or low fidelity, prototyping (Kelley, 1984), while another is high-fidelity prototyping using a computer-based tool.

Wizard-of-oz studies are widely used for analyzing the effectiveness of a low-fidelity user interface. According to Höysniemi et al. (2004), the first researchers to use the technique were Gould et al. (1983) and the term itself was coined by Kelley (1984)\(^1\). The basic idea of the approach is to use simple mock ups and prototypes with a human behind the scenes manipulating them.

Usability studies often play a different role than paradigms like the one described in this chapter. Usability studies, both low and high fidelity, are typically used when you want to know how participants might try to use a program. However, the often unrestricted nature of usability studies can make measuring something as specific as comprehension difficult to cleanly separate.

Artifact encoding gives us an objective comprehension measure outside the context of a particular program. Artifact encoding does not, however, tell the researcher when to use an auditory cue, only the level to which it will convey its intended meaning. Simply put, artifact encoding studies help determine comprehension issues (see Chapter 5), while usability studies put auditory cues into context (see Chapter 7).

\(^1\)Technically, Kelley called the technique “Oz,” not “Wizard of Oz,” but the name “Wizard of Oz” has persisted in the literature.
Auditory Cue Creation

The wizard-of-oz technique is powerful, as it allows the researcher to conduct usability studies on a potential user interface without having to implement it. However, conducting wizard-of-oz prototyping studies on auditory displays has significant downfalls. If a researcher is using non-speech audio, as is extremely common in the literature, including amongst others, Rouben & Terveen (2007); Sporka et al. (2006); Stifelman (1995); Alty & Rigas (1998, 2005); Hankinson & Edwards (1999); McGookin & Brewster (2004); Blattner et al. (1989); Brewster et al. (1993, 1996); Brewster (2003); Brown et al. (2006), the procedure by which examples are created can prove difficult (Vickers, 1999; Stefik et al., 2006b).

For example, if a researcher is using musical examples, physically creating the MIDI for the usability tests can be very time consuming, depending both on the complexity of the music and what the music represents. In my previous work (Stefik et al., 2006b), musical examples were so complex that it took several months to create them by hand, which can hardly be considered low fidelity. In other words, in auditory interfaces, sometimes the time to create the audio by hand outweighs the time taken to build the interface that will generate the audio. Study designers should carefully weigh the time it will take to complete their auditory cues. Sometimes it is faster to build the tool first.

4.1.3 Software Engineering Metrics

Another approach to analyzing the auditory cues of an audio based program execution environment is to map software engineering metrics to participants’ ability to comprehend auditory cues. Probably the most significant work that uses this approach was done by Vickers & Alty (2002b, 2003, 2005).

The idea of a Vickers style empirical study is to use a software engineering metric, in his case program faults and participants’ debugging ability, as a gauge for the effectiveness of the auditory cues. Program faults, or more colloquially “bugs,” is one of many metrics that can be used to
measure how well auditory cues are working for a particular cue.

While software metrics can give one idea of how well auditory cues might perform in a real world task, it is difficult to determine why an auditory cue had an effect on a given metric without knowing something about how participants understood the auditory cues. For example, in the case of using bugs as a measure of auditory cue efficacy, we know from prior research that bug types vary in frequency and difficulty (Spohrer & Soloway, 1986), and correlative studies are needed to determine whether certain types of bugs are related to the effectiveness of the audio. It is likely that certain types of auditory cues reveal certain types of program faults more than others.

Thus, even if a software engineering metric is chosen well, it is difficult to say whether particular metrics are representative for particular types of auditory cues without knowing something about the comprehension of those cues. In other words, studies that involve complex real world procedures (e.g., bugs found, dollar cost or person hours to complete a programming project) have a number of unidentified independent variables which implicitly exist, threatening internal and construct validity.

4.1.4 Discussion

There is a natural trade-off in empirical work between internal and external validity. As experiments become more realistic, they become increasingly reflective of reality but less internally consistent and harder to control. Multiple choice studies and artifact encoding both err on the side of internal validity, being tightly controlled, but are not necessarily reflective of real world performance. Contrasting multiple choice and artifact encoding, multiple choice studies are simpler to design, but are data poor and give only a rough approximation of users’ comprehension. Artifact encoding gives a thorough record of comprehension and can be automatically graded.

More realistic studies (e.g., usability studies or those based on software engineer metrics), are often too complex to measure something like comprehension directly. Studying issues like what type of cues to use for scoping would be incredibly difficult in a usability study, as there are far
too many features, besides scoping, that can influence a dependent variable. On the other hand, the comprehension of one individual auditory cue might have little or no effect on a real world task, and as such, has a low effect size for the average user. By using both tightly consistent and real world tasks, we can test the minutia of an environment in a highly consistent way (like scoping: see Chapter 5), while also testing an environment versus the state-of-the-art (SOD vs. Visual Studio.NET 2005 + JAWS 9: see Chapter 7).

4.2 Artifact Coding

Artifact encoding is an experimental paradigm that tries to determine how well participants understand a set of auditory cues. At its simplest, artifact encoding works by first giving participants some form of stimuli, like auditory cues, and then asking participants to interpret the meaning of those cues. The mismatch between participants’ interpretation and the meaning the researcher intended can then be calculated, which gives us an objective measure of whether the auditory cues were interpreted as expected. In this section, I will describe artifact encoding in detail.

4.2.1 Example: Animal Identification

As a first example of artifact encoding, consider participants identifying animals in photographs. Participants might see a finite set of pictures, one every 10 seconds, including a dog, a cat, a fish, or an elephant. In this case, a participant would be asked to identify, for example by writing on paper, the type of object they are seeing, keeping a record of the entire sequence.

Suppose the participant is given $n$ pictures, where $k$ animals are possible at each position, and that each animal is shown to participants $n/k$ times in a random order. Data from this procedure can then be collected from a series of participants and analyzed for patterns. If, for example, the picture of a cat is often misidentified, this implies participants had more difficulty identifying the photograph of that animal.

In the case of the identification of animals, we can detect issues like order or context effects in addition to identification. For example, consider a study where pictures were shown that had
various characteristics: color quality, graininess, or other attributes. Perhaps grainy cats were
misidentified more often after seeing pictures of low color fish. While these attributes hold little
meaning in terms of pictures of animals, order matters in computer code.

Consider an auditory cue for the statement $a = b;$, perhaps “a equals b semicolon.” This
reading requires participant to understand that the character $=$ literally means to assign to. Un-
fortunately, the idea that $=$ means assign is not always intuitive. McIver (1996, 2000, 2001), for
example, uses the left arrow, $\leftarrow$, to indicate assignment in her programming language GRAIL,
which was designed to have a more intuitive syntax.

4.2.2 Procedure

Artifact encoding studies are broken into a series of three phases: training, identification, and
interpretation. In training, participants are given information on how to complete the study. Iden-
tification determines whether participants correctly heard the right cues, and interpretation tries to
determine if participants understood the meaning of the cues.

The first phase in artifact encoding is an explanation and training phase. Using audio to perform
programming tasks may be a new, and to some users, unexplored venue. As such, experimenters
need to be wary of a novelty effect (Shadish et al., 2002). In pilot studies, I have experimented
with using varying levels of training for audio based tasks.

How much training participants need depends on the type of auditory cues used in the study.
For example, in my experience, the amount of training required for music-based auditory cues
appears to be greater than for speech-based cues. This result accords with the literature, as musical
based earcons have been shown to be more difficult to learn than their compressed speech based
spearcon counterparts by nearly 700% (Palladino & Walker, 2007). Further, the idea that speech
or other auditory cues might be easier to learn is far from surprising. Participants have a lifetime
of learning to interpret speech, but this is not the case with musical cues.
The second phase in artifact encoding is *identification*. Participants listen to a stream of auditory cues, which they subsequently write down either verbatim or through a shorthand. During this phase of the experiment, participants are not asked to interpret the information in the auditory cues.

The final phase in artifact encoding is *interpretation*. In this phase, participants analyze their previously recorded auditory cues and translate them into marked-up computer code. The answers participants give in this phase are coded for analysis. One might immediately guess that, given that participants in this phase are translating from work in a previous phase, the identification step can be skipped and participants can interpret immediately after hearing the audio. I attempted using this approach in a pilot study (Stefik et al., 2007) and found that it has major problems.

First, if the identification step is skipped, participants will need to hold an entire audio example in memory while completing the task, which for long auditory cues is not feasible. Second, while identification could potentially be skipped for speech based auditory cues, for musical cues (Brewster et al., 1993; Vickers & Alty, 2005), or auditory icons (Gaver, 1986), identification could be very informative for determining how well a participant can disambiguate a given set of sounds. Including the identification step allows a user to write down their own descriptions of the sounds they are hearing. This is important, because if a number of participants cannot find adequate ways to describe the sounds, those sounds may be ineffective as tools for auditory display.

In sum, the procedure for conducting artifact encoding is broken into three primary phases: training, identification, and interpretation. The training phase usually comes before identification and interpretation, but identification and interpretation are often inter-mixed. Thus, in a typical study, participants are first trained, after which they complete a series of identification-interpretation, pairs.
if ( T ) {}

or

if ( F ) {}

Figure 4.3: This shows the notation participants use to represent a true or a false if statement.

4.2.3 Notation

In an artifact encoding study, participants are asked to listen to auditory cues, interpret them, and then write down their interpretation of those cues. Asking participants to complete free-form writing tasks can make interpreting participants’ answers difficult, and as such, participants are asked to use a standard notation and format to represent their answers. In this section, I describe the notation used in an artifact encoding study.

Let us begin the analysis of notation by looking at Figure 4.3. This figure shows the notation used by participants to represent if statements. First, notice that the expressions participants write are minimal. In this thesis, I am predominately testing behavioral runtime cues, and as such, participants cannot know the expression inside of an if statement, only the behavior induced by the expression’s result. The left and right curly braces are optional. Many participants opt not to include them in their answers.

The statement “if ( F )” actually means what is shown in Figure 4.4. When conducting pilot studies on the notation, I discovered very quickly that participants did not like writing out the additional curly braces, and as such, the shorter “if ( F )” was allowed as a shorthand. Identifying minor compromises like this are important for an artifact encoding study, as the tasks are sometimes tedious and difficult. The notation should not make these tasks significantly more so.

When notating an if statement, the participant writes the dynamic behavior of the if by indicating either T or F, true or false. With loops, however, the participant needs to indicate the behavior of the code on each iteration of a loop. Figure 4.5 shows an example of a loop that iterates twice. On the first iteration of a loop, the if statement inside of it evaluates to true, while
if ( \text{F} ) 
\{
\}
else //code executed here 
\ldots
\}

Figure 4.4: This notation shows what “if ( F )” really means in practice, that the code executed into an else block.

\begin{verbatim}
while ( 2 ) {
  if ( \text{T}, \text{F} ) 
  {} 
}
\end{verbatim}

Figure 4.5: In this example, the 2 indicates the number of iterations of the loop. On the inside of the loop, the first T indicates what happened on the first iteration, while F indicates what happened on the second iteration.

A more complex example of the notation for a loop is given in Figure 4.6. In this example, there are two if statements inside of the loop, which are serial, not nested. The notation here indicates that the loop executed twice. The first if statement evaluated to true on the first iteration and false on the second, while the second if statement did the opposite.

Thus, the notation in an artifact encoding study enables participants to write down, at least when behavioral runtime cues are in use, both the static structure of the code and the dynamic behavior of the code. Examples of the notation become considerably more complex as more statements are added to the code. In the next section, I expand this notation further, while also discussing how I translate the notation into coded answers that can be graded automatically by a computer program.

\begin{verbatim}
while ( 2 ) {
  if ( \text{T}, \text{F} ) 
  {} 
  if ( \text{F}, \text{T} ) 
  {} 
}
\end{verbatim}

Figure 4.6: In this more complex example of a loop, there are actually two if statements. The first if statement evaluated to true on the first iteration and false on the second. The second if statement did the opposite.
Table 4.2: A table of different sounds used in formal experiments, their intended meanings, and how they are coded.

### 4.2.4 Coding

Once participants write their answers to a set of artifact encoding problems, data are coded into a standard form and graded. In this section, the process of translating participant answers into a coding system is described. A thorough set of examples of this coding process is given in Appendix A.

Table 4.2 gives a list of the codes I have adopted for translating participants’ notations into a format that can be graded. This table shows (a) what the code means, (b) an example sound, and (c) an answer that would reveal the code. Notice that the notation discussed previously is translated into a string of characters, which correspond to the structure and behavior of the computer code.

Consider the situation in which a participant writes down an `if` statement. If we use behavioral runtime cues to represent an `if` statement, then our cue might be either “if true” or “if false.” The goal of the coding system is to represent that the participant heard an `if` statement and its truth value. As such, I represent the identification of the `if` as the letter I, and true or false as either T or F.

Coding an `if` statement is simple enough in isolation, but we also need to be able to code multiple `if` statements, which are either serial or nested. Consider Figure 4.7, which would be coded as ITNIFU. In this case, the N and the U act as curly braces, `{}`, in a C-like programming language. The IT part of the code indicates an `if` statement that evaluated to true and N indicates to increase the nesting level. Similarly, the IF code indicates there was an `if` statement that evaluated
if (T) {
  if (F) {
  }
}

Figure 4.7: This participant’s answer would be coded as ITNIFU.

int i = 0;
while (i > 1) {
}

Figure 4.8: This is an example of code where a loop exists, but never actually executes. This would be correctly coded as a single L.

to false, and the U indicates that the nesting level should be decreased. The false if statement has nothing inside it, and therefore the coding is not ITN IFNU U. In other words, even though the participant indicated curly braces for the false if statement in Figure 4.7, they are not included in the coded answer.

Figure 4.9 shows auditory cues that would indicate that a single loop iterated four times. If a participant notated these auditory cues correctly, their answer would look like Figure 4.10. The coding for the answer given in Figure 4.10 would be LNU LNU LNU LNU \(^2\), or four LNU triplets. In this case, unlike if statements, if there is nothing in the scope of the loop, the NU pair is used on every iteration. This is because a sole L has a special designation, meaning that a loop existed in the code, but never executed. Figure 4.8 gives an example where this might be the case. Thus, while LNU four times indicates a loop executed four times, LLLL implies four independent loops that never executed.

\(^2\)Technically, the answer is LNULNULNULNU. Spaces are added only to make the answers easier to read.

loop iteration 1,
loop iteration 2,
loop iteration 3,
loop iteration 4,
end loop

Figure 4.9: This is the auditory cue that would be heard for a loop executing four times.
while(4) {
}

Figure 4.10: The coding for this answer is be LNU LNU LNU LNU.

loop iteration 1,
1 nested if true,
2 nested if false,
loop iteration 2,
1 nested if false,
2 nested if true, end loop

Figure 4.11: This is the auditory cue for a more complex loop, which iterates twice, each time with different behavior.

A more complex example of auditory cues is shown in Figure 4.11. Its corresponding answer is shown in Figure 4.12. In this example, the final coding is LN ITNIFU U LN IFNITU U, with spaces separating loop iterations for ease of reading. The first part of this coding is LN, which indicates the first iteration of the loop is beginning. The second clump of coding is ITNIFU, which indicates there was an if statement that evaluated to true at runtime, with a nested if statement inside of it which evaluated to false. The last part of the first loop is a single U, which indicates the end of the first iteration of the loop.

The next iteration of the loop, LN IFNITU U, starts similarly, but then has an if statement which evaluates to false. Since the participant’s answer implies that the next if statement is still nested, this means that the while loop has an else statement attached to the if.

Since artifact encoding notation can be occasionally confusing, Figure 4.13 represents the same dynamic trace of a program as Figure 4.12 slightly differently. In the case of Figure 4.13, the underscores represent where control flow does not reach. For example, on the first iteration of the loop, the if statement was true, and as such, control flow entered into the true portion of the if. On the second iteration of the loop, however, the statement was false, and as such entered into the else portion of the if.

The character Z is used to indicate when a participant made a mistake that no other coding
while(2) {
    if(T,F) {
        if(F, T){}
    }
}

Figure 4.12: The coding for this answer is be LN ITNIFU U LN IFNITU U. Spaces were added between iterations of a loop to make this coding easier to read.

while(2) {
    if(T,F) {
        if(F, _){}
    } else {
        if( _,T) {} }
}

Figure 4.13: This example is the same as Figure 4.12, although it is represented differently to show where control flow goes in the program more easily. The underscores here represent regions of the code that control flow does not reach.

for the example could easily represent. An example is shown in Figure 4.14, where a participant wrote a nonsensical answer where two if statements, attached by an else, both evaluated to true. Suppose the actual answer to the problem was two serial if statements, both true. This would mean the correct answer is ITIT. To indicate the mistake with the else, however, we add a Z character, making the final answer ITZIT. As we shall see in the next section, the basic result of coding the participants answer in this way is approximately equivalent to docking them one point.

Lastly, while the codes and notation provided here are used for the experiment in Chapter 5, alternative codes and alternative notations can be used without altering the study paradigm. The

```
if ( T ) {
}
else if ( T ) {} {}
```

Figure 4.14: An example where the Z character is used in coding. The coding here would be ITZIT.
Figure 4.15: A basic overview of Artifact Encoding, including the automatic grading process.

Process of adding a code (e.g., assignment statements, switch statements, functions) involves modifying the automating grading algorithms, a trivial process.

4.2.5 Measures and Grading

Participant grading in an artifact encoding study is accomplished by comparing sets of coded strings. Strings are first aligned globally using techniques from the amino acid sequence comparison literature (Needleman & Wunsch, 1970). Second, strings are post-processed to gather information about how participants understood a running computer program. This process is demonstrated in Figure 4.15.

Global Alignment

Consider the very simple participant answer, correct answer, pair given in Table 4.3. In this case, the participant appeared to realize that two if statements executed, but did not realize the second
if statement was nested inside the first.

The first part of the grading process is to globally align the strings. Comparing small strings, as in Table 4.3, is relatively trivial. With longer examples, however, it becomes increasingly more difficult to compare the strings manually. As such, unlike previous work (Stefik et al., 2007), once strings are coded, all further analysis of the data is accomplished by the Artifact Encoding Calculator (AEC – pronounced “ack”), shown in Figure 4.16.

Using amino acid sequence comparison algorithms (Needleman & Wunsch, 1970), we can obtain what is called a global alignment between two strings. A global alignment is basically a match between the two strings. The Needleman & Wunsch (1970) algorithm is guaranteed to find the optimum match between the strings, via whatever criteria you give the algorithm for calculating what optimum means. There are, however, two minor differences in the typical Needleman & Wunsch (1970) implementation and my own.

The first difference is that, in Needleman & Wunsch (1970), the language of acceptable characters in the algorithm is defined as the legal characters used for DNA, \( \sum = \{ A, G, C, T \} \). In my artifact encoding study, however, I define \( \sum = \{ I, T, F, N, U, L, Z \} \). This difference is minor and has no bearing on the effectiveness of the algorithm.

Second, in Needleman & Wunsch (1970), optimum is usually defined by saying how matched two particular characters are within a string. In other words, the character T, if matched to another T, might be worth 5 points. Similarly, N matching with N might be worth 3 points. In other words,
characters can be weighted to give a better match than others. In my case, however, almost all characters are given an equal weight, with one exception. In if statements, we want to be able to detect whether participants determined the correct truth value for a statement, or alternatively, whether they “flipped” a boolean.

To see why this might be a problem, consider globally aligning two strings ITNITU and ITIF. If all characters in the string are equally weighted, then it would be equally acceptable to match the string ITIF as either IT-IF- or IT-I-F, where dashes indicate a gap when compared to the string ITNITU. However, it would be desirable for the Needleman-Wunsch algorithm to automatically match up the string as IT-IF-, not IT-I-F, because then we can iterate through the string looking for these flipped booleans. Fortunately, accomplishing this is easy. By assigning all same characters as a weight of two points, T and F a weight of 1 point, and no match at all a weight of zero points, our strings will be globally aligned in the desired way. Once we have aligned the strings, we can then move on to picking out comprehension attributes.

Figure 4.16: A Screenshot of the artifact encoding automatic grading program, using a modification of the Needleman-Wunsch algorithm Needleman & Wunsch (1970).
Comprehension Attributes

A global alignment, in and of itself, does not tell us useful information about the comprehension of auditory cues. For that, we need to post process globally aligned strings. Specifically, we might want to know how well the participant did overall, how well the participant understood scoping, or how well the participant understood scoping at various scoping levels. We might also want to know whether the participant flipped any booleans, or how often the participant left $\textit{if}$ statements or loops out of an answer.

Figure 4.16 shows an example of two strings being matched and then analyzed post hoc. In the textbox labeled String 1, the original answer to the problem is given, whereas the textbox labeled String 2 is a participant’s answer. The results from Needleman & Wunsch (1970), and a series of post-processing metrics, are shown across the bottom.

The first metric of interest is the comprehension score, a global measure of how well the participant correctly interpreted the auditory cues. To calculate the comprehension score, we sum the number of matched pairs in the two strings. For example, given the string ITNITU and ITIF, where the matched strings are ITNITU and IT-IF-, the comprehension score would be $4 / 6$ or $66\%$. In Figure 4.16, a much more complicated alignment and comprehension score is calculated, where the result is $54.25\%$.

Besides the comprehension score, Figure 4.16 demonstrates four more metrics. The first metric is incorrect scoping. To calculate this metric, consider the simple example of ITNITU and ITIT. To calculate this measure, we linearly post-process both strings in parallel. Whenever both strings are at an I or an L character, we check to see if the scoping level of both constructs is equal, which we know by counting the N and U characters that are within the string.

If both I and L characters in the second string are at the same scoping level as the first, nothing happens, but if they are incorrect, we add one point to the number of incorrect scoping constructs at the appropriate scoping level. There is a problem, however: what if we have the strings, ITNITUU and ITITNITU. In this case, using the previous algorithm, we will count both the second
and third if statements at the wrong scoping level, even though the participant must have known that the third statement was scoped inside the second, as shown in Figure 4.17.

The type of problem, demonstrated by Figure 4.17, is an example of a cascading error. Removing this type of error is important. If it is not removed, one mistake at the beginning of an answer will cascade through the rest of the answer. Fortunately, removing this type of error is easy. Suppose two characters are compared at a given scoping level and found to be incorrect. If this happens, the participant’s answer is artificially set to the correct scoping level and processing is allowed to continue. The practical result of this is that participants are only docked points for every individual statement that is incorrect within the local scope, as opposed to being docked globally.

I define three additional metrics for measuring auditory cue comprehension: flipped booleans, missing if statements, and missing loops. A flipped boolean is counted whenever a T is matched with an F in the globally aligned strings. We know this will be a good match, since Needleman & Wunsch (1970) was altered to align these characters appropriately. Missing if statements and missing loops are calculated similarly. Whenever the participant string contains a gap, but the answer string contains an I or an L character, a missing if statement or missing loop is accordingly counted.

It would be tedious to manually enter each string into the Artifact Encoding Calculator shown in Figure 4.16. Thus, the Artifact Encoding Calculator reads an entire dataset from Microsoft Excel. A new spreadsheet is then output to the user with all of the pertinent results tabulated.
4.3 Conclusion

In order to build non-sighted interfaces from the ground up, this chapter has presented a method for measuring the effectiveness of auditory cues called artifact encoding. There are three major benefits of the artifact encoding paradigm: *comprehension metrics, natural mistakes, and automatic grading.*

Artifact encoding provides a suite of metrics related to human comprehension of auditory cues: comprehension, incorrect scoping, flipped booleans, missing `if` statements, and missing loops. Each metric represents one type of issue a human might have when trying to interpret a cue, offering insight into a participant’s comprehension and comprehension process.

Multiple choice studies rely on determining comprehension mistakes a priori (predetermined forced choice). In contrast, artifact encoding allows users to interpret auditory cues as they see fit. This is a significant advantage, as multiple choice studies can only reveal comprehension mistakes that the researchers were able to guess might be a problem before a study was conducted, whereas *artifact encoding demonstrates the comprehension mistakes people actually make.*

Automatic grading may seem a nicety of artifact encoding, but, in fact, it is critical to the paradigm. Take, for example, the study presented in the next chapter. Each participant generated approximately 500 data points. These points are separated into a set of strings, which are processed by Needleman-Wunsch, and then are processed for relevant information about how the participant comprehended a program.

If a researcher were to accomplish this by hand, given 200 participants, they would need to find a global alignment between 10 strings, for 10 tasks, per participant. Ensuring that participants answers are matched up to actual answers in the correct way requires creating an $n \times m$ matrix for each task for each participant. If we assume approximately 50 characters per string, then this means the researcher must perform $50 \times 50 \times 10 \times 200 = 5,000,000$ calculations, a slow and boring proposition. Further, these initial calculations only provide global alignment! In short, automated
techniques are essential for performing artifact encoding studies.

Artifact encoding provides a method for analyzing the comprehension of auditory cues apart from any particular user interface. Usability studies and software engineering studies (top down evaluation methods) are often too complex to pick up on the subtleties of human comprehension. In contrast, artifact encoding provides an analysis method for analyzing auditory cues from the bottom up. In the next chapter, I use artifact encoding to study the effect of adding scoping cues to an existing set of behavioral runtime cues.
CHAPTER 5

EMPIRICAL STUDY: AUDITORY CUES FOR CONVEYING SCOPE

Certain classes of programmers, especially the visually impaired, need to acquire information about computer programs using either audio or haptic stimuli. Creating effective stimuli is an important endeavor because the ability of individuals to acquire such information is critically dependent upon the concept or construct being learned, the way in which the construct is expressed, and the mode in which it is delivered to the individual.

The purpose of this empirical study is to determine the effectiveness of different expressions of audio based stimuli for representing both the relationships between individual program constructs and the constructs themselves. Program constructs in computer code are highly interwoven and related. Variables in assignment statements impact program behavior and the results of a statement can be carried over to the next statement. Program constructs within the scope of other constructs can have their meaning affected, or determine whether they execute, depending upon the exact relationship with their parent constructs.

In the context of visual interfaces, surface, superficial, and redundant sensory cues reinforce the relationships among program constructs (e.g., source code formatting). To my knowledge, no researcher has yet created and verified equivalent, effective, cues for programming paradigms using audio only stimuli, nor has a researcher analyzed the effect of scoping cues on auditory program comprehension. In terms of the design space presented in Chapter 3, scoping cues are a form of localization cue, meaning they inform the user about static relationships between program constructs.

But why focus on scoping cues? There are two primary reasons for focusing on this type of cue: effective scoping cues help the non-sighted determine their context in a program (Where am I?), and scoping cues offers a good initial test of the artifact encoding paradigm. When non-sighted programmers listen to auditory cues, they need to determine their textual location—and how that
position relates to the rest of the source code. Effective scoping cues help with this type of task by answering context questions (e.g., Is this loop inside or outside the if statement I recently heard?).

In addition, scoping cues provide a good initial test of the artifact encoding paradigm. Consider the possible results of the study. If the results indicate that adding my scoping cues have no effect for any of the empirical measures, it implies that either the cue did not alter a programmer’s comprehension or that the paradigm is not sensitive enough to detect these effects. In contrast, if significant differences are detected, this implies that artifact encoding was able to detect the differences between the auditory cues—validating the experimental paradigm’s ability to hone in on comprehension issues.

5.1 Methods

5.1.1 Design

Across a set of ten tasks, participants were asked to listen to speech based sounds and interpret the runtime behavior of a running computer program. Artifact Encoding was used, and as such, participants were first trained, then they completed a series of ten identification/interpretation, pairs. Participants were randomly assigned to one of two groups related to the auditory cues used in that group. The total design, however, was $2 \times 2$ with two independent variables: scoping auditory cues and programming experience. My hypothesis is that scoping cues will significantly decrease the number of scoping errors and that experience level will have an effect on the comprehension of the auditory cues.

The scoping independent variable had two levels, either with or without scoping auditory cues. The auditory cues heard by both groups were identical except for the added scoping cue. The experience independent variable also had two levels. Specifically, some participants were enrolled in a programming course with no prerequisites, while others were enrolled in a course that required the successful completion of at least one previous programming course.

A number of dependent variables were used to determine the effect of adding scoping cues on
the total comprehension of the auditory cues. There were 5 objective measures of comprehension: *a total aggregate comprehension score, the percent of program constructs placed at the correct scoping level, flipped booleans, and missing loops and if statements.* These measures, how they are calculated, and their meaning, are discussed in detail in Chapter 4.

While my hypothesis is most concerned with the metric that indicates the number of scoping cues correctly placed at the appropriate scoping level, poor results for other comprehension metrics (e.g., flipped booleans, missing data) might indicate that adding scoping cues has side effects. In other words, while the scoping cue correctness metric indicates scoping accuracy, if the results indicate that other comprehension metrics are negatively impacted, this could imply that a different type of scoping cue is desirable.

5.1.2 Participants

Sixty-four sighted participants were recruited from the computer science program at Central Washington University for this study. As an incentive, students were given approximately 3% extra credit toward their coursework. Since recruiting large numbers of non-sighted programmers, at various experience levels, was not practical, I instead used sighted proxies.

Out of the 64 participants in the study, 45 were male and 19 were female. Participants ranged in age from 18 to 48 years old, mean 22.8. Students self reported having from 0 to 10 years of computer programming experience, mean 1.9 years, and were recruited from every class in the computer science department. Participants ranged from non-major freshman taking Visual Basic to seniors in the software engineering and advanced algorithms courses. Specifically, 39 of the students were in their first programming course, while 25 of them had already completed at least one course on programming. Table 5.1 shows the number of participants in each cell of the $2 \times 2$ design.

While the teaching language at Central Washington University is Java, participants reported experience in a variety of languages. Specifically, participants reported having approximately 0.75
years experience in C++, 1.07 years in Java, 0.52 years in basic, 0.253 years in C#, and a mean of 0.417 years experience writing HTML code. While some participants reported experience in other programming languages, the overall means were negligible.

5.1.3 Materials and Tasks

Participants completed a series of ten experimental tasks, their response to which was recorded in a provided booklet. Auditory cues were output from stereo speakers that were sitting in approximately the middle of a room. The volume of the speakers was loud enough for all participants to hear (as indicated by an exit questionnaire), and the volume was kept constant between experimental conditions.

The stimuli used in this study was a stream of speech based audio that related to a running computer program. Using the categorization of auditory cues from Chapter 3, the cues in use were behavioral runtime cues either with or without indicators related to the scope of a program construct. The computer programs consisted largely of conditional statements and loops at various levels of scoping. Care was taken to make the programs semantically neutral, meaning that the participants could not determine the purpose of the program from the auditory cues. In other words, the auditory cues were designed to tell the user nothing about whether the program was a banking application, a video game, or any other specific application.

The sounds for the audio were created using the Sonified Omniscient Debugger (SOD), a program execution environment that will be discussed further in Chapters 6 and 7. Speech was read by Microsoft’s text to speech engine, using the voice Microsoft Anna—English (United States). In order to facilitate understanding of the audio based stimuli, I carefully tested the speed at which the running computer program was read. Adjusting this using Microsoft’s text to speech engine

<table>
<thead>
<tr>
<th>Previous Programming classes</th>
<th>Scoping Cues</th>
<th>No Scoping Cues</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>1 or More</td>
<td>13</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.1: This table shows the number of participants in each cell of the 2x2 design.
is straightforward, as a slider bar can be adjusted in the operating system to change the reading speed. In the case of Microsoft Anna, the speed chosen was two clicks below the label “Normal” in Microsoft’s text to speech properties box.

As for the experimental tasks themselves, each task consisted of computer code. The code included either executing loops or if statements, the auditory cues of which were presented to the participant. Tasks were designed randomly, with some constructs being nested, some not, and loops executing any number of times. Each task was unique and therefore had unique auditory cues. Figure 5.1 shows the code used by the Sonified Omniscient Debugger to generate the auditory cues for task number six.

5.1.4 Procedure

This study was conducted in a quiet room at Central Washington University over the course of two days. Each experiment required approximately an hour and a half to conduct, making it impractical to run all 64 participants through the experiment individually. Thus, participants were assigned to
<table>
<thead>
<tr>
<th>Time</th>
<th>Tuesday</th>
<th>Wednesday</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:00 AM</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>10:00 AM</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1:00 PM</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>3:00 PM</td>
<td>14</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5.2: This table shows the number of students that participated at each time of the study.

<table>
<thead>
<tr>
<th>Time</th>
<th>Tuesday</th>
<th>Wednesday</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:00 AM</td>
<td>No Scoping</td>
<td>Scoping</td>
</tr>
<tr>
<td>10:00 AM</td>
<td>Scoping</td>
<td>No Scoping</td>
</tr>
<tr>
<td>1:00 PM</td>
<td>No Scoping</td>
<td>Scoping</td>
</tr>
<tr>
<td>3:00 PM</td>
<td>Scoping</td>
<td>No Scoping</td>
</tr>
</tbody>
</table>

Table 5.3: This table shows which group was assigned at which time for the experiment.

blocks of time. Each day there were four blocks. Participants were asked to sign up for a given time and day, but were told neither the hypothesis of the experiment nor their experimental group.

While every time block would ideally have exactly the same number of participants, in practice the 8 AM time slot was unpopular and the afternoon sessions tended to have more students. Table 5.2 shows the number of students in each session. Students were randomly assigned to groups as well as possible.

This study was also conducted double blind. Washington State University paid a proctor, who neither knew the hypothesis of the study nor was a computer scientist, to run all sessions with students. This proctor was carefully trained on how to run the experimental sessions, was given a script for how to interact with students, and had standard answers for many common questions students might have.

In order to assign students to groups, the time of day that a group would receive stimuli for either the scoping cues group or the no scoping cues group was randomly assigned. On the second day, the times for each session were reversed. Table 5.3 shows what groups were assigned to what times for the experiment.
Lastly, in order to prevent cheating, cardboard spacers were placed between all students. Students of any height would have to stand up and look over their own, and another student’s cardboard spacer in order to cheat. The study proctor did not report any incidents of cheating during the course of the experiment.

*Training Procedures*

Participants in both groups went through an identical, three-phase training process: *passive listening*, *answer preparation*, and *practice problems*. During passive listening, participants listened to a set of program constructs without taking action. This part of the training process acclimated participants to how auditory cues can be used for understanding running computer programs. Participants passively listened to examples of true `if` statements, false `if` statements, nested `if` statements, loops, and loops with embedded `if` statements. Figure 5.2 shows an example from the passive listening portion of the training, including the explanatory text from the experimental packet.

While Figure 5.2 shows an example that the group with scoping cues received, the non-scoping cues group received almost identical cues. The only difference was that the last line indicated the auditory cue for that experimental group, namely: loop iteration 1, if true, loop iteration 2, if false, end loop. Notice that in this case, although the scoping cue was removed, the participant could still determine that the `if` statement was inside of the loop, as the fact that the loop eventually terminated, signified by the words “end loop,” implicitly encoded the concept of scope for this particular construct. Figure 5.3 shows another example of code and its corresponding auditory cues for both groups.

Each example during passive listening was first played one time from a compact disc. This compact disc included the text to speech generated for the program executions and also included a studio recorded script, which read an explanation to the participants. The voice actor that recorded the speech did not know the hypothesis of the study. After playing an example one time, the
int main() {
    int i = 0, k = 0;
    while(i < 2) {//execution begins here
        k++;
        if(i == 0) {
            k++;
        } else {
            k--;
        }
        i++;
    }
    return 0;
}

Figure 5.2: An example from the passive listening portion of the training during this study. The caption in the study states: “The sounds in this example are the same as those before, but in this case, an IF statement is nested within a while loop. This loop executes twice. During the first iteration of the loop the IF statement evaluates to TRUE and during the second iteration it evaluates to FALSE. The final sound produced by this execution sequence is LOOP ITERATION 1, 1 NESTED IF TRUE, LOOP ITERATION 2, 1 NESTED IF FALSE, END LOOP.”

if( a == b ) {
    if ( b == c ) {
    }
}

Figure 5.3: This is a simple example of a nested if statement. The caption in the study states: “The auditory cue, assuming both statements evaluated to true, would be IF TRUE, 1 NESTED IF TRUE for the scoping group, while for the no scoping group it would be IF TRUE IF TRUE.”
while( 2 ) {
    if( T,F ) {
    }
    else {
    }
}

Figure 5.4: This example is from the answer preparation phase of the experiment. The caption in the study states: “The number two in the while loop means the body of the loop executed twice, while the letters T,F indicate that the first time the loop executed the if statement evaluated to true, but that it evaluated to false the second time.”

caption for the example was spoken via the voice actor’s recorded speech. Once the caption was read, the example was played a second time, and the training moved on to the next example in passive listening.

Once the passive listening phase of training was complete, participants entered the answer preparation stage. During answer preparation, participants were taught how to notate their answers. Participants answers came in two forms: identification and interpretation. During identification, participants wrote a shorthand that represented an auditory cue. During interpretation, participants looked at their shorthand and attempted to determine the static structure, along with the dynamic behavior, of the program. Participants were initially taught how to accomplish these tasks by performing copy tasks, where a correct version of a final answer was given, without audio, and this information was copied from one location to another.

Determining an adequate notation for participants’ final answers required extensive pilot testing to calibrate. The primary difficulty was in integrating runtime behavior information with a static representation of code. Figure 5.4 gives an example of how this information is represented. The number two in the while loop means the body of the loop executed twice, while the letters T,F indicate that the first time the loop executed the if statement evaluated to true and that it evaluated to false the second time.

The final phase of training was the practice problem phase. During this phase, participants were given small examples relevant to the actual study. This phase mimicked the tests used in the
study, except: the examples were much smaller, the sounds were played twice instead of once, and
the final answers were given after the participants attempted them.

While this three-pronged procedure for training may sound complex and time consuming, the
entire process only took about 20 minutes. Further, giving participants an initial idea how to write
their answers and use sound for understanding computer programs helped (a) remove potential
novelty effects, (b) improved the consistency of participants’ answers, and (c) gave participants
practical experience applying the training to practice problems.

Task Procedures
For each task, participants first listened to the auditory cues, at which point they wrote down a
shorthand notation of what they were hearing. This notation was used for two reasons. First, during
pilot testing, participants, regardless of what you told them to write during the identification phase,
wrote some kind of shorthand anyway, and as such, standardizing what they wrote made answers
more consistent. Second, if participants could not write down what they heard before interpreting,
they would have to remember several minutes worth of auditory cues in their head. This would be
difficult, if not impossible, for the complex auditory cues used in this study.

After the participant heard the auditory cues, a recorded voice indicated to the participant that
it was time to interpret what they wrote. During this process, which lasted three minutes for each
task, participants translated their shorthand into the final notation described earlier. Last, once
participants had completed all ten experimental tasks, they took a ten minute exit survey.

5.2 Results

5.2.1 Reliability Analysis
Once the data was complete, participants’ answers were coded into strings. To ensure that this
encoding process was reliable, I conducted a Kappa analysis (Dewey, 1983; Hubert, 1977). Two
researchers trained on data not used in the current study, and then independently coded approx-
imately 20% of the actual data. A Kappa statistic of 0.904 (raw agreement 92.07%) was found,
which is considered very high. Appendix A describes the coding process in detail.

After the data was coded, string representations of participants’ answers were sent to, and automatically, graded by, the computer. This was accomplished by using a combination of amino acid sequence comparison algorithms (Needleman & Wunsch, 1970), and post processing algorithms, as discussed in Chapter 4. This technique produced a spreadsheet with the results embedded. Statistical analysis was then completed on the data in this spreadsheet.

### 5.2.2 Performance Measures

Univariate ANOVAs were used for the statistical analysis. For all measures, the interaction effect between scoping cues and experience level was non-significant, and as such, these results will not be mentioned further. Table 5.4 summarizes the results.

![Table 5.4: A summary table of the experimental results](image)

Figure 5.5 shows the results of the total comprehension metric with scoping cues as the independent variable. The comprehension metric represents an aggregate of how well the participant performed. How to calculate this value is complex, and is described in detail in Chapter 4, but a series of algorithms were used to take into account whether participants correctly identified *if* statements, loops, iterations of loops, scoping, and whether *if* statements were true or false. Figure 5.6 shows the same metric with experience on the x-axis.
Figure 5.5: This figure shows the percent correct for the aggregate artifact encoding score, called comprehension, on the y-axis. On the x-axis is the independent variable of scoping cues, either without them (left), or with them (right).

Figure 5.6: This figure shows the percent correct for the aggregate artifact encoding score, comprehension, on the y-axis. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).
The difference between those who received scoping cues ($M = 62.53$, $SD = 25.8$) and those who did not ($M = 52.26$, $SD = 16.8$), approached significance, $F(1, 60) = 3.952, p = .051$. The effect size of this value was modest (Partial Eta Squared = .062). In contrast, the difference between those with experience ($M = 64.69$, $SD = 20.9$), and those without ($M = 52.72$, $SD = 22.0$) was significant $F(1, 60) = 4.52, p = .038$. This effect was also modest in size (Partial Eta Squared = .070). As one might expect, participants with little programming experience had difficulty understanding computer code using audio stimuli.

Two measures of missing data were extracted from the experiment. The effects for the scoping cue difference was shown in Figure 5.7, and for experience is shown in Figure 5.8. There were two measures here: missing if statements and missing loops. A count of 1 point means that that there was one construct, either a loop or an if, that the participant did not write down in his or her answer (summed from all ten tasks).

Those that received scoping cues ($M = 8.81$, $SD = 9.7$) had more missing loops than those that
Figure 5.8: This figure shows the number of missing loops and if statements, summed from all ten tasks, on the y-axis. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

did not (M = 5.16, SD = 8.35), but not significantly so, \( F(1, 60) = 2.750, p = .102 \). Similarly, those that received scoping cues (M = 25.13, SD = 29.9) had more missing if statements than those that did not (M = 18.97, SD = 31.5), although this result was also non-significant, \( F(1, 60) = .574, p = .451 \). It is interesting to note that that the scoping cues group performed worse under both measures, although neither was significant.

If we consider just those who had a prior programming course (M = 3.12, SD = 3.29), we find a significant difference between the two conditions, no experience (M = 9.46, SD = 10.8), with respect to the “missing loops” measure, \( F(1, 60) = 8.521, p = .005 \). The same also held true for if statements, with the experienced group (M = 9.60, SD = 10.5) and the not experienced (M = 30.03, SD = 36.31), also being significantly different \( F(1, 60) = 7.486, p = .008 \). The effect size of both loops (Partial Eta Squared = .124), and if statements (Partial Eta Squared = .111) was moderate in size.

Figure 5.9 Shows the percent of program constructs, loops or if statements, placed at the
Figure 5.9: This figure shows the percent of program constructs, loops or if statements, that were placed at the correct scoping level. On the x-axis is the independent variable of scoping cues, either without them (left), or with them (right).

correct scoping level, either with or without receiving scoping cues. Figure 5.10 shows the same metric, but with experience level on the x-axis.

The mean percentage of program constructs placed at the correct scoping level for the group that received scoping cues (M = 79.8, SD = 13.1), was significantly higher than the no scoping cues group (M = 59.2, SD = 10.1), $F(1, 60) = 54.474, p < .001$. The effect size was quite large (Partial Eta Squared = .476). However, the group that had no experience (M = 69.0, SD = 14.35) was not significantly different from the group with experience (M = 70.3, SD = 17.5), $F(1, 60) = .041, p = .841$.

The last objective metric of interest, flipped booleans, measures how many times participants stated that an executing if statement evaluated to false, when it actually evaluated to true, or vice versa. Figure 5.11 shows the flipped boolean measure with whether the participant received scoping cues on the x-axis, while Figure 5.12 shows the same metric but with experience on the x-axis.
Figure 5.10: This figure shows the percent of program constructs, loops or if statements, that were placed at the correct scoping level. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

Figure 5.11: This figure shows the number of flipped booleans in participants’ answers. On the x-axis is the independent variable of scoping cues, either without them (left), or with them (right).
Figure 5.12: This figure shows the number of flipped booleans in participants’ answers. On the x-axis is the independent variable of experience, whether the previous number of programming classes participants had taken was either 0 (left), or 1 or more (right).

The group that received scoping cues (M = 7.6, SD = 8.1), had more flipped booleans than the group that did not receive scoping cues (M = 5.4, SD = 4.7), although this result was not significant $F(1, 60) = 2.193, p = .144$. Those participants with experience (M = 6.2, SD = 8.3), had less flipped booleans than those that did not (M = 6.7, SD = 5.4), but this effect was also non-significant, $F(1, 60) = .103, p = .750$.

5.2.3 Survey Results

In addition to the objective measures mentioned previously in the results section, participants were asked, after completing the study, to fill out a small survey about the tasks they completed. Survey questions were given on a 5-point Likert scale. When asked if the sounds in SOD were intuitive, the group that received scoping cues rated the overall sounds more intuitive (M = 3.59, SD = 0.98), than those that did not (M = 2.97, SD = 1.43), a difference that was statistically significant, $F(1, 60) = 4.417, p = .040$, (Partial Eta Squared = .069).
Participants were also asked to rate the speed of the auditory cues. A rating of 5 meant too fast, 3 was about right, and 1 was too slow. The scoping cues group suggested that the cues were too fast (M = 4.00, SD = .76), while those that did not receive the cues suggested the speed was closer to “about right,” (M = 3.19, SD = 1.06). This effect was also significant, $F(1, 60) = 11.892, p = .001$, (Partial Eta Squared = .165).

The scoping cues group found loops less helpful for determining scoping relationships (M = 2.47, SD = 1.19), than did those without scoping cues (M = 3.28, SD = 1.19), $F(1, 60) = 6.643, p = .012$, (Partial Eta Squared = .100). Similarly, those with experience (M = 3.28, SD = 0.98) found loops more helpful than those without experience (M = 2.61, SD = 1.35), $F(1, 60) = 5.428, p = .023$, (Partial Eta Squared = .083). The group that did not receive scoping cues felt that loops helped them determine scoping level more than the scoping cues group, probably because the scoping cues group already had specially embedded cues for dealing with that issue. In addition, the group that was enrolled in their first programming course thought loops were not as helpful as those in a higher programming course. This is not surprising, as the low experience group had been introduced to loops for the first time only a few days prior to participating in the experiment.

The last significant results from the follow up survey indicated that participants with little experience felt they did not have enough training (M = 1.56, SD = .94), as compared to those with more experience (M = 2.44, SD = 1.08), $F(1, 60) = 12.124, p = .001$, (Partial Eta Squared = .168). Similarly, participants with little experience (M = 2.49, SD = 1.26) felt they understood the tasks worse than those with more programming experience (M = 3.76, SD = 1.05), $F(1, 60) = 17.474, p < .001$, (Partial Eta Squared = .226).

5.3 Discussion

There are several observations to be made about the previous results. First and foremost, scoping cues that state the nesting level explicitly, followed by the the program construct, (e.g., 1 nested if true) increase a participant’s ability to understand the scope of a program construct. This appears
Figure 5.13: This figure shows the percent correct for the comprehension metric for every task, to hold across experience level.

Second, scoping cues did not cause any negative side effects that could be significantly detected in this study. While the group that received scoping cues did have slightly higher rates of missing data (especially loops) and did have a slightly higher incidence of flipped booleans, these trends were not significant. It should be noted, however, that while those who received scoping cues made fewer scoping errors, that even with these cues, participants were able to obtain only about 80% accuracy. Other auditory cues may be able to increase this percentage.

Third, scoping cues do not universally offer better aggregate comprehension of auditory cues, as seen in the list of tasks in Figure 5.13 Notice that in some cases, the two groups are approximately equal, and in the case of tasks 1 and 8, the no scoping cues group actually outperforms the scoping cues group, although not significantly so. Similarly, Figure 5.14 shows the effect of scoping cues on each task.

While participants made fewer scoping errors for all tasks in this experiment, qualitative inspection of participants’ data suggests reasons for task differences. For example, it is easy to determine the scoping relationship between any individual if statement if it is the first if statement inside a
loop. The auditory cue for a loop, even if it only executes one time, was, “Loop iteration x.” The cue “End loop” indicates when the conditional inside the loop has evaluated to false. This implies that for certain loops, with certain structures, one can determine the scope of the construct without needing a scoping cue.

This works for loops because in a debugger the loop eventually has to evaluate to false, unless it is an infinite loop. But if statements do not repeat, making the end of the construct difficult to determine unless the closing brace of a statement is sonified. However, data from the experiment suggests that participants were able to determine the scoping level, even without the closing brace sonified, approximately 80% of the time. This means that the current scoping cues effectively allow the user to identify scoping information without having to browse up and down the source looking for a closing brace.

An analysis was also conducted on scoping cues at various levels of nesting. Level one is the top-most scoping level; it would appear “flush left” if one were looking at a computer screen. In this case, we see that the scoping cues peak at level one, then dip. The dip is lowest at around level 3. The no scoping cues group appears to do the opposite, peaking in its effectiveness around level 3.

Figure 5.14: This figure shows the percent correct for scoping in each task.
Figure 5.15: This figure shows the percent of correctly placed program constructs at each level of scope.

3. I hypothesize that, for the no scoping cues group this is caused predominately by loops. For example, at scoping level 1, participants should be able to determine that constructs that come after the beginning of the loop, but before its end, are at the same scope. The reasons for the dip between level 1 and 2 in the scoping cues groups, however, is not entirely clear.

5.3.1 Qualitative Analysis

Participants made a series of interesting mistakes during the course of the experiment, which will now be discussed qualitatively. There were two major mistakes of interest: one related to loops, and another related to a particular misinterpretation of the auditory scoping cues used in this study.

First, consider the auditory cue given in Figure 5.16. Most participants interpreted this auditory cue as meaning that a loop, with no if statements or other significant control flow inside of it, executed twice. This was the intended meaning of the construct. When a participant wanted to indicate a loop that executed twice, they were told to write this down as shown in Figure 5.17.
Loop iteration 1
Loop iteration 2.

Figure 5.16: An example auditory cue representing a loop that executes twice and has no significant control flow within it.

```c
while(2) {
}
```

Figure 5.17: The correct written answer for the auditory cue listed in Figure 5.16.

Notice that the 2 indicates the number of iterations, while the loop itself it just an empty body.

Some participants, however, misinterpreted the iteration number to indicate not when the loop executed, but instead the loop’s scoping level. The most typical way this was misrepresented is shown in Figure 5.18. Notice that the participant thought that the loop iteration 2 statement, if you take the marks literally, indicated a second loop was inside the first one. This mistake was relatively rare, but interesting because it illustrates one way that this particular auditory cue can be misinterpreted, which gives us clues on how to improve the cue. This mistake was usually made by those in the no experience group.

A better looping auditory cue should make it obvious that the words Loop Iteration 1 and Loop Iteration 2 identify the same loop. One possibility might be to add an identifier to loops, perhaps with “Loop A Iteration 1, Loop A Iteration 2,” but this might have consequences in making the participant think the letter A indicates the conditional inside the loop. Another possibility would be changing the prosodic component of a cue for every unique loop. For example, the fundamental frequency of the text to speech engine’s voice could change for individual loops. Alternatively, a non-speech sound, musical or otherwise, could be embedded to play in conjunction with loops, and this sound could be unique for individual loops.

A final example of a common mistake made by participants relates to the auditory cue, “3 nested if true.” While most participants were able to correctly interpret this cue as meaning an `if` statement that is scoped within two other program constructs, approximately 10% of the Scoping Cues group misinterpreted this cue to mean that there were `three if statements at the same scoping`
while(1) {
    while(2) {
    }
}

Figure 5.18: A relatively uncommon misinterpretation of the auditory cue listed in Figure 5.16. This mistake was usually made by those registered for their first computer programming course.

    if true
    if true
    if true

Figure 5.19: Some participants misinterpreted the auditory cue 3 nested if true to mean three if statements that executed, all with the same truth value.

level.

Figure 5.19 gives an example of what a participant might write down for the auditory cue in question. Thus, for example, if the auditory cue said, “4 nested” the participant would write down 4 if statements, all with the same truth value. This misinterpretation is interesting for several reasons. First, participants thought that the auditory cues were grouping together like constructs. While no concepts are grouped in the current auditory cues, the prospect of doing so is actually quite compelling, as minimizing the number of auditory cues is important, and grouping program constructs intelligently could provide an interesting type of auditory glance (Stevens, 1996).

In addition, this mistake suggests a relatively straightforward way of fixing the incorrect interpretation. One possibility for doing so would be to move the “x nesting” part of the cue to the end of the construct, instead of having it at the beginning. As such, “3 nested if true” becomes, “if true 3 nested.” Another possibility is to use prosodic cues, in addition to word choice and placement, to fix the misinterpretation. In this case, text to speech engines could be asked to emphasize the words “3” and “if.” Or, alternatively, a pause could be inserted between the words nested and if, which would prevent the participant from hearing the cue as meaning, “3 nested if” then “true,” and would instead force them to hear, “3 nested,” then “if true.”
5.3.2 Threats to Validity

It is clear that scoping cues do increase a participant’s ability to understand scoping relationships in a computer program. It is also clear that scoping cues do not cause negative side effects, nor are the cues effectiveness relative to experience level. However, do comprehension measures of the kind presented here hold any practical external validity? That is—if a comprehension score is lower, does that mean a programmer will be less effective for comprehension and debugging tasks?

While more evidence is needed, the answer appears to be yes. In Chapter 7, I discuss the auditory cues used in Visual Studio ©2005 using the Jaws ©9 screen reader. In this study, the Visual Studio cues, which output the words “F9” whenever a key is pressed in the debugger, would have an artifact encoding comprehension score of zero (e.g., F9, F9, F9, F9 does not tell the user important information regarding the execution of a computer program). This means programmers using this environment must rely on other auditory cues (e.g., the raw syntax), to prime their mental model of a computer program.

As I will demonstrate in Chapter 7, poorly understood auditory cues cause programmers to complete comprehension and debugging tasks significantly slower than when using auditory cues with a higher comprehension score. This result makes sense: it is hard to program, manipulate, or debug what one does not understand.

5.4 Conclusion

In this chapter, I have presented an empirical study that used artifact encoding to measure a human’s comprehension of a set of auditory cues. The results of this study indicate that the particular scoping cues I designed were effective indicators of scoping and that their inclusion did not negatively impact the other objective comprehension metrics presented here. In addition, I have identified several ways participants misunderstood the auditory cues and have proposed potential ways to improve them.

In the next chapter, I discuss the Sonified Omniscient Debugger (SOD), a program execution
and debugging environment that implements the auditory cues used in this chapter. After the
discussion of SOD (see Chapter 6), I will compare the auditory cues presented here with those of
Visual Studio ©2005 using the Jaws ©9 screen reader (see Chapter 7).
CHAPTER 6
DESIGNING AN AUDIO BASED PROGRAM EXECUTION ENVIRONMENT

Designers of environments for non-sighted computer programmers need to decide what information to present to users, what auditory cues will represent that information, and under what context those auditory cues will be presented. Non-sighted computer users currently obtain this information, and hear these cues, using screen readers that connect to third party applications. These third party applications are retrofitted with audio; they are not built with non-sighted users in mind.

One way in which retrofitted applications are insensitive to the needs of the non-sighted community is that they ignore the fact that users of auditory interfaces must heavily rely on working memory (Baddeley, 1992). This is due, in part, because auditory interfaces, especially those involving speech, are naturally serial in nature. Visual users have less demanding working memory requirements due to their ability to scan a computer screen.

In contrast to retrofitted tools, the Sonified Omniscient Debugger (SOD), a program execution and debugging environment designed explicitly for non-sighted users, was built from the ground up. This environment includes cues that were either empirically analyzed in comprehension studies or that were designed to be sensitive to the working memory demands of the non-sighted community. In this chapter, I discuss my ground approach to building SOD’s user interface (audio/visual) and architecture.

6.1 SOD: The Sonified Omniscient Debugger

In Chapter 3, I gave an overview of various types of auditory cues: design, editing, or execution. In Chapters 4 and 5, I analyzed a selection of these cues with artifact encoding, focusing specifically on comprehension issues related to execution cues. However, in order to test whether these cues lead to usability improvements in real comprehension or debugging tasks they must be integrated
into a prototype program execution environment.

My prototype environment, The Sonified Omniscient Debugger (SOD), is both an omniscient debugger (Lewis & Ducassé, 2003) and a sonified debugger. In an Omniscient Debugger, the goal is to make the debugging environment know everything about the running program. This implies storing all possible state changes as a program executes, which allows the user to step a program both forward and backward. Besides being omniscient, SOD is also a sonified debugger. This means that any information from the compiler can be given to the user via some form of auditory cue.

As an example of the advantages of a debugger based on the philosophy of omniscience, consider programming in a typical C/C++ environment. If a program were to crash, as often happens while developing in C++, the programmer would need to restart the program and set a breakpoint to the location they suspect caused the crash. In SOD, however, programs do not crash. Instead, SOD verbally informs the user that a program would have crashed (and why). In the next two sections, we will explore both SOD’s visual and aural interface, including how it presents omniscient information to its user.

6.1.1 Visual User Interface

Figure 6.1 shows the basic SOD visual interface while a C program is executing. The primary window in SOD has syntax highlighted source code, a toolbar with typical features, like building and executing, and a menu system. The tabbed windows below the main source window contains compiler related messages, like syntax errors, and variable information similar to a local variables window.

SOD also includes features like opening and saving files, undo and redo, creating new files, executing programs, breakpoints, etc. Users can compile source in the usual manner, although unlike typical C programs, a pre-processor is not included, nor is every syntactic element in the C language implemented.
When the user begins debugging using the current interface, the program is first placed at the beginning of its execution sequence. The user can then step forward and backward in the source either on a line-by-line basis or by jumping ahead, or back, to a breakpoint. Executing a program in SOD is similar to using an undo/redo list in a program. While the user interface itself works on a line-by-line basis, the compiler executes at the semantic level, meaning that the internals of the compiler have nothing to do with the lines and columns in the code. This makes integrating auditory cues into the compiler easier.

6.1.2 Auditory User Interface

While the auditory interface in SOD is complex, and provides many more cues than execution cues, behavioral run time cues have undergone the most extensive empirical testing (see Chapters 5 and 7). Thus, in this section on the auditory interface in SOD, I focus on the behavioral runtime cues presented when a program is executed. Each part of this section focuses on the different sounds that are made when a particular type of statement is executed (e.g., an expression, if
if( a == b ) {
}

Figure 6.2: Assuming variables a and b were equal, this statement would be read, “if true.” The compiler automatically determines that the value of a == b should be spoken as either true or false when in the context of an if statement.

int c = a == b;

Figure 6.3: Assuming variables a and b were equal, this statement would be read, “set c to 1” The compiler in this case does not assume that the value of 1 is supposed to mean either true or false.

Simple expressions

There appears to be two primary choices for representing a simple comparison expression (e.g., a == b in a C-like language), using a behavioral runtime cue. The behavior of the statement is that it either evaluated to true or false, so we can either say true/false or 1/0. While using 1 and 0 might appeal to a C programmer, there seems to be little reason to force the user to remember the fact that 1 maps to true and 0 maps to false.

Further, since modern programming languages do away with the arbitrary 1 or 0 mapping anyway, the current choice for a mapping in SOD is to literally state the value of “true” for a non-zero value in an expression and “false” for a value of 0. Unfortunately, this mapping does not always work. Recall that in C, given an expression like a == b, one cannot detect whether the expression should indicate the numeric value of the statement or if it is symbolically indicating true or false. For example, consider the statement, int i = 0; . When this statement executes, the value placed into the variable i will be given the value of zero. It's possible that the programmer will use the value of this variable as an indicator for true or false, but there is no way to know in an isolated statement.

As such, in the SOD compiler architecture, the true or false constructs are stated only if the expression lies in the context of an if statement or a while loop. As an example, Figure 6.2 shows the auditory cue for an expression within an if statement, whereas Figure 6.3 shows the
exact same expression outside of a construct.

Assignment statements with primitives

To represent runtime information for assignment statements, the most obvious choice is to read the value of the assignment. For primitive values this process is easy. Take, for example, the statement, \( a = b + 1; \). In this statement, we can state the value of the assignment, (e.g., “a equals 5,” “set a to 5,” or “a set to 5”) In SOD, the cue of choice is, “set variable a to 5.”

Floating point values, however, like \( 3.14159265 \), take time for a screen reader to say, and the user may not actually need the entire floating point value. For example, it seems reasonable to think that the user will likely only need the first few digits of a floating point value, except in special situations. Unfortunately, research trying to determine an adequate number of digits is still anecdotal (Kildal & Brewster, 2007a), and applications are still at the pilot study stage of development (Kildal & Brewster, 2007b). As such, SOD reads the entire value, but allows the user to skip the end of the auditory cue using either the arrow keys or the function keys (e.g., F9 for step over).

Assignment statements with composite structures

When objects, structs, or other composites are used in assignment statements it becomes less obvious what the auditory cue should be. Consider a composite \textit{bank} that is being assigned the value of an instantiated bank class at runtime. In a language like Java, where the syntax would be, \texttt{Bank} \texttt{bank} = \texttt{new Bank();} it might make sense, at runtime, to say, “bank is assigned a new Bank” or “set bank to new bank.” This tells the user that a bank object was instantiated and that this value was put in memory at the location of the variable bank, but tells little else. This representation of the assignment does not include information like default values or initial behavior, which a user might be interested in.

Further, what if we assign an existing instantiation of a Bank to the variable bank? In this case, \texttt{bank = someExistingBank}, we might say, “bank is assigned an existing instantiation
if ( a == b ) {
    if ( c == d ) {
    }
}

Figure 6.4: A set of nested if statements.

of the class Bank,” or “set bank to someExistingBank.” These auditory cues are not very helpful, however, as they give the same information as a static read of the code.

Another option would be to read “set bank to bank object id 4875,” for some object (or instantiation of a struct) with a particular id number 4875. This would at least tell users which bank object is being assigned, which would allow them to query the object further if they wish. Unfortunately not all programming languages support object features in this manner, including the programming language C, the language of choice for this thesis. As such, for structs in C or C++, the value given to the user would likely be a raw memory location, the utility of which is unclear.

Conditional statements

It is less difficult to determine a reasonable auditory cue for if statements than it is their corresponding expressions. if statements, in SOD, are indicated at runtime literally by the word, “if,” which is followed by the truth value of the expression within that if statement. Testing has shown this choice of words to be quite effective (See Chapter 5).

Scoping

Every program construct has a corresponding scope. An if statement can be nested within another if statement or a loop. The first option for an auditory cue is to not indicate the relative scoping of program constructs. Consider the very simple example given in Figure 6.4. Suppose that this computer program is currently executing at the line, if ( c == d ) {. Assuming, for example, that c == d were true, if we state this line of code as “if true” then the user is left guessing as to the relationships between the current and previous program construct.

A behavioral runtime cue of Figure 6.4 should attempt to indicate to the user the scope of both
if true {
    if true {
        if true {
        }
    }
}

Figure 6.5: An incorrect interpretation of “3 nested if true.”

if( a == b ) {
    if( c == d ) {
        if( e == f ) {
        }
        if( e == f ) {
        }
    }
}

Figure 6.6: A more complex set of nested if statements. One auditory cue related to scoping might indicate scoping depth, like “1 nested if true,” while another might indicate the beginning and ending of the if statements using the words “nest,” “unnest,” or something similar.

if statements. Suppose additional words are added to the statement, like “1 nested if true” to indicate that it is the first nested construct within a previous if. However, does this auditory cue indicate scoping relationships, despite the fact that a closing brace is not indicated? We know from Chapter 5 that the answer is yes.

In contrast, there are other approaches for sonifying nested constructs in computer code besides using the “one nested” cue. For example, we could use a phrase like “nest if true unnest.” As an example, consider the code presented in Figure 6.6. Using the first scoping technique, this section of code would read, “if true, 1 nested if true, 2 nested if true, 2 nested if true,” assuming all if statements evaluate to true at runtime. Instead, if we use the second technique the cue could be, “if true, nest, if true, nest, if true, if true, unnest, unnest.” Words like “left brace” and “right brace” could also be substituted for nest and unnest. SOD uses the “1 nested” style of cue because it does not require users to navigate the source code looking for an end brace to determine scoping information and empirical evidence suggests that it is effective.
while( a == b ) {
   a++;
}

Figure 6.7: A loop

**Loops**

Consider the loop presented in Figure 6.7. To determine appropriate auditory cues one must first consider what type of operations a loop performs and which of those operations a user might care about. Similarly to if statements, loops have two possible conditions, either a loop evaluates to true, executing one branch of code, or executes false, thereby executing an alternate branch. Unlike if statements, however, if a loop’s conditional is true, control is returned to the conditional and the process repeats after the true branch of code executes.

Users may want to know several characteristics about a loop’s behavior. Some possibilities include if the loop’s current expression evaluated to true or false, how many times the loop executed, which iteration of a loop is currently being executed, and when the loop ends. Unlike if statements, however, the sonified debugger does not indicate whether the result of the expression in a while loop is true or false.

Instead, when a loop evaluates to true for the first time, SOD will say, “loop iteration 1.” If loops are nested, like in Figure 6.8, the scoping indicator, “n nested” like before, is used. There are several important attributes to notice about this auditory cue. First, the term “loop” is used instead of the term “while.” Previous empirical studies on sonification using both musical feedback and speech based feedback showed that some types of loops, especially for loops, can sometimes be confused with the number four (Stefik et al., 2006b,a; Begel & Graham, 2006, 2005). Thus, all loop types are indicated with the word “loop” to avoid the issue.¹ More evidence is needed to determine if using the word loop for any kind of iterative structure (e.g., for, while, do-while), is a comprehension issue for behavioral runtime cues.

¹I suspect this choice would be problematic for editing cues but not for execution cues.
while( a == b ) {
    while( c == d ) {
        c++;
    }
    a++;
}

Figure 6.8: A nested loop

A second consideration in loops is that there are two numbers being presented to the user when loops are nested. Consider the beginning of an auditory cue for the code presented in Figure 6.8. This auditory cue would begin, “loop iteration 1, 1 nested loop iteration 1.” In the case of the nested loop, the first number, 1, indicates the nesting level, while the second number indicates the current iteration of the loop.

The next important decision in loops is whether iteration numbers should begin at loop iteration 0 or 1. While this is a contentious issue, I found in pilot studies that having a zero based index caused more off by one errors. \(^2\)

Lastly, note that nowhere in the auditory cue for a loop are the words true or false used. This is because if a loop is iterating, it must be the case that its conditional evaluated to true, otherwise a cue indicating the end of the loop would be played. Essentially, this means that the truth value of a loop’s conditional is redundant information and does not need to be presented to the user. In general, if empirical evidence indicating a good auditory cue is unavailable, *pick an auditory cue that provides the most information in the least amount of audio.*

*Functions*

I preface this section by noting that the cues presented for functions are partially hypothetical, as SOD only partially implements C functions. Consider the simple C program presented in Figure 6.9. On line 2 of the call in `main`, the function `myFunc()` is called. When this occurs, SOD will say the words, “calling method myFunc.” While some language designers prefer to make a

\(^2\)Some participants argued that a zero based indexing was superior. These same participants, however, made plenty of off by one errors.
int myFunc() {
    return 5;
}

int main() {
    myFunc();
    return 0;
}

Figure 6.9: A simple function, myFunc, being called from a main method in C.

int main() {
    int result;
    result = myFunc();
    return 0;
}

int myFunc() {
    return 5;
}

Figure 6.10: A function that is called as part of an expression will be aurally hidden, telling the user the value of the expression but not indicating that a function was called. In this case, the line, `result = myFunc();` would be sonified as, “set variable result to 5.”

distinction between the word “function” and “method” the SOD architecture makes no distinction.

Return values from functions are considered expressions and they are sonified as discussed previously. In this case, SOD will say, “returning 5” when executing the statement `return 5;`.

When a function ends it states “ending method myFunc.” Similarly, the debugger could end the function by saying, “returning 5 from myFunc.”

If the user decides not to step into the method myFunc, and instead steps over, the sound is different. If the function is called like in Figure 6.9 the result will be, “calling method myFunc” with no other attributes. There is no need to indicate the return value in this case, as the value will be thrown away by the compiler. In contrast, if the return value is used as part of an expression, the result of the function call will be hidden to the user as part of a larger expression.
If a function includes recursion, the auditory cues could mimic the sounds of loops, indicating the number of consecutive times a function has called itself, either through direct or indirect recursion. It remains to be seen whether recursive depth is helpful to a user.

6.1.3 Discussion

Current programming interfaces for the non-sighted are retrofitted with screen reader support, but many of these interfaces seem to have barely considered, if at all, what types of sounds might actually be useful in the interface. Further, this area has been only scantly touched in the literature. I find it significantly odd that, after decades of work on visual interfaces, that sound’s potential role in programming has been given so little investigative effort.

As can be seen from the previous section, there is an immense number of choices to be made in regards to non-sighted programming interfaces. Some of these choices can be made with little more than common sense, while others appear to have trade-offs or potentially problematic side effects (Does “2 nested if true” really indicate to the user that the source code at that position is within two constructs, or will users interpret this as meaning two if statements?) While I feel these sections have only scratched the surface of using sound in programming, I now move on to the more technical details of how SOD is constructed.

6.2 Architecture

In order to build an environment for the non-sighted, including the auditory cues previously discussed, the Sonified Omniscient Debugger was built from scratch. SOD includes several primary components, namely an interpreter, a debugger, and several typical features seen in modern development environments. In this section, I discuss the technical details of the components and their interactions.

The traditional approach to compiler design is to break the total functionality of the compiler into a series of phases. Appel (2002) describes twelve phases in typical compiler development: lex, parse, semantic actions, semantic analysis, frame layout, translate, canonicalize, instruction
selection, control flow analysis, data flow analysis, register allocation, and code emission. While these phases will not be described in any detail here, the core idea, with any compiler, is to read the text into a computer friendly form, lex and parse, create an abstract syntax tree representing the source, and then give meaning to the program elements via semantic analysis. Further compiler phases handle details of putting the program into an executable form or optimizing the code.

The custom interpreter in SOD is created in Java and parses the C programming language. The parser was created in Javacc, a relatively modern LL(k) parsing environment, and was marked up using JJTree. JJTree is a tool that marks up a Javacc grammar with information in order to automatically create an abstract syntax tree representation of the source code. An open source C grammar was used, originally written by South (1997), but was heavily modified for use with JJtree and SOD.

Figure 6.11 shows the primary components of the SOD architecture. In the remainder of this section, I will discuss the technical details of the SOD architecture and the relation between the user interface, the compiler, the debugger, and the sound system.

In order to make SOD reasonably language neutral, the parser and virtual machine are wrapped into a set of abstractions that allow other virtual machines for other languages to be substituted. The SonifiedDebugger2 component stores the main program logic for SOD. This class has three components: HighlightedDocument, UndoManager, and AbstractVirtualMachine.

HighlightedDocument is an open source Java based syntax highlighter created by Ostermiller (2002). While SOD is a sound based tool, it includes many visual features (e.g., syntax highlighting, visual debugging cues, a highlighted yellow line at the current point in execution, toolbar buttons with reasonable graphics and pictures). Adding these features makes it possible to test auditory environments versus equivalent visual environments, as is done in Chapter 7.

The UndoManager is Java’s solution for undoing and redoing elements of a document that is being edited by the user. The implementation in SOD of undo and redo is typical. Interestingly, AbstractVirtualMachine objects execute programs in a remarkably similar fashion.
Figure 6.11: Overview of the SOD architecture
The AbstractVirtualMachine houses the primary, language neutral, model for executing within the environment. Any AbstractVirtualMachine object can take a string representation of a source file, or a set of files, and build an appropriate program. Regardless of the implemented programming language, the virtual machine can step or unstep across the current execution sequence, essentially equating to running the program one step forward, one step backward, or running the program under a set of conditions.

The execution model of SOD uses a visitor and listener, called a VirtualMachineListener, for instrumenting programs. VirtualMachineEvent objects are thrown by any given AbstractVirtualMachine implementation, informing the user interface of that machine’s current state.

This process allows the user interface to determine the current state of the VM (e.g., what line and column the ExecutionStep is on), and also allows listeners to determine whether they want to do anything on that step. As an example, consider the internal compiler event of storing temporary memory for variables in an expression. If the sound based debugger were given a listener that cared about these types of events, they could be sonified at runtime, giving aural output to a user. Indeed, I found this type of listener hierarchy extremely useful for debugging the compiler itself.

The VirtualMachineC object subclasses the AbstractVirtualMachine object for the C programming language. The VirtualMachineC class delegates its operations to three primary components: CParser, SymbolTableVisitor, and VirtualMachineStateC. The CParser parses the source code sent by the virtual machine and, if there are no syntax errors, creates an abstract syntax tree in the usual manner.

The SymbolTableVisitor object contains visit methods, where the parameter is every class in the abstract syntax tree in the C programming language. It uses these methods to build a C program, caching pertinent information along the way. This class does semantic analysis on the source code and determines whether the source code is compilable. If the class is not compilable, it adds CompilerError messages to an instance of a CompilerErrorManager, which the user interface will later be able to access in a VirtualMachineEvent message. If, however, the program succeeds in
compiling, the SymbolTableVisitor will make a series of ExecutionStep objects, in conjunction with the ExecutionBuilder, and eventually create a vector of indexed steps the program will go through to execute.

The Execution class handles the logic of executing a compiled program. The Execution class is also language neutral, meaning a virtual machine, for any implemented language, can use it to execute programs. The Execution class contains an aggregate of compiled ExecutionStep objects, each of which tracks its current state and previous state. The ExecutionManager knows, through the ExecutionStep objects, how to step forward in the program, but unfortunately, since any arbitrary point in execution can be jumped to from any other arbitrary point, a stack of ExecutionStep objects is stored as the program executes in order to reverse execution.

Last but not least, any environment for the non-sighted requires special libraries for outputting sound. To do this, the AbstractCStepVisitor class contains methods (more than are shown in Figure 6.11) for every type of ExecutionStep object implemented in SOD. In order to create auditory cues that correspond to the runtime behavior of a program, a VirtualMachineListener is created. This listener then delegates ExecutionStep objects to a hierarchy of AbstractCStepVisitor objects, which define what auditory cues are played to the user.

6.3 Conclusion

In this chapter, I have discussed the design and implementation of an audio based program execution and debugging environment, the Sonified Omniscient Debugger (SOD). SOD is a virtual machine for the C programming language designed to facilitate the study of programming environments for non-sighted programmers, including the design of appropriate auditory cues.

Many challenges remain in the design of audio based integrated development environments. The SOD architecture considers predominantly behavioral runtime cues, and some editing cues, but does not integrate other auditory cues or tools that may help users. In the next chapter, I test the cues designed through artifact encoding (see Chapters 4 and 5), in a summative evaluation to
see how SOD compares to a state-of-the-art development environment and screen reader.
CHAPTER 7

EMPIRICAL STUDY: SCREEN READER VS. SOD

In chapter 5, I empirically explored the comprehensibility of behavioral runtime cues, but did not evaluate whether greater comprehension, as defined by those measures, constituted a real improvement over existing screen readers. In this chapter, I address this issue by evaluating SOD in realistic comprehension and debugging tasks.

The reasoning behind having two types of empirical studies in this thesis has to do with the natural trade off between internal and external validity. In artifact encoding studies, participants are given auditory cues and asked to interpret their meaning. The method and procedure for measuring this interpretation is tightly controlled, making the experimental tasks less realistic but internally consistent and easy to interpret.

But does the comprehension of auditory cues actually matter when writing or debugging software? When programmers write software, they work within an interface (e.g., The command line, an integrated development environment, a text editor). While these interfaces may have constraints built in, programmers are not as controlled as they would be in a typical psychological experiment. Programming tasks are messy, something which an artifact encoding study cannot take into account without sacrificing the reliability of the measures.

To evaluate the effectiveness of the SOD environment in more realistic tasks, I experimentally compared it against (a) a visual environment (used to establish an upper bound on human performance) and (b) a current state-of-the-art environment for the non-sighted: Microsoft ® Visual Studio ® 2005 coupled with the JAWS ® 9 screen reader. In this study, I found that the SOD environment promoted significantly more efficient task performance than did Visual Studio coupled with JAWS. Moreover, whereas a significant difference was found between the visual environment and Visual Studio + JAWS, no such difference was found between the visual environment and SOD. These results indicate that SOD constitutes a significant improvement over the current
7.1 Sample Session with SOD

In order to provide a feel for how the SOD environment is being used in this experiment, I now walk through a sample session in which I use the SOD environment to complete a portion of one of the tasks performed by participants in this experimental evaluation (see Section 7.2.1). The task is to answer a comprehension question about the “Replace Values” algorithm, which replaces values in an array that are less than 25 with zero. For comparison purposes, I describe in this example both SOD’s auditory cues, and the state-of-the-art auditory cues that are generated by the JAWS 9 screen reader coupled with Microsoft Visual Studio 2005.

Figure 7.1 presents a visual snapshot of the SOD environment. The C source code for the same buggy “Replace Values” algorithm used in my experiment is loaded into the environment. In this example, I consider how a user might answer the comprehension question, “How many items have been replaced after the third iteration of the loop?” The user could answer this question using three different methods. Illustrating these three methods will expose us to SOD’s repertoire of auditory cues.

In the first method, which is the most difficult, the user would first place the source code into working memory, and then mentally simulate the algorithm to arrive at an answer. To do this, the user would listen to the source code by pressing the up and down arrow keys, which cause the environment to play cues corresponding to the line being visited. Whereas pressing the arrow keys produces semantic cues in SOD, it produces syntax cues in Visual Studio + JAWS. For example, when visiting the statement \( m[3] = 14; \), a SOD user would hear “m sub 3 equals 14.” In contrast, a Visual Studio + JAWS user would hear “m left bracket three right bracket equals fourteen semicolon.”

The second method the user could employ would be (a) to execute the loop three times by repeatedly pressing the F9 key to forward execute each line of code, and then (b) to count the
number of times the assignment statement that replaces array values (line 13 in Figure 7.1) is executed. To illustrate how this would be done, let us assume that the user is positioned at the while statement highlighted in Figure 7.1. From there, the user would need to press F9 four times in order to navigate through one complete loop iteration.

On the first F9 press, the user would hear “loop iteration one” (line 11). On the second F9 press, the user would hear “one nested if true” (line 12). This cue conveys two pieces of information: (a) that the if statement is nested inside the while loop, and (b) that the if statement evaluated to true, since the value at v[0] is, in fact, less than or equal to 25, as can be seen in the Locals Window. By pressing the F9 key a third time, the user would hear the cue “set array v sub 0 to 0” (line 13), indicating that an array element had been assigned a value. An alert user would also recognize this statement as a bug, since the value of k will not always be 0, as required; however, I will not get into the details of fixing the bug in this example.

Hitting F9 a fourth time, the user would hear the loop increment statement on line 15: “set
variable k to one.” Finally, on the fifth press of F9, the user would begin the second iteration of the loop (“loop iteration two”), at which point the entire process would start over.

Contrast the previous set of auditory cues to those generated by Visual Studio + JAWS. Each time F9 is pressed, the Visual Studio + JAWS user would hear “F9.” In order to hear information regarding the line that was navigated to, the user would have to press the up arrow key followed by the down arrow key. For example, if the user pressed F9 to execute the \texttt{if} statement on line 12, and then hit the up and down arrow keys, the user would hear the cue “graphic 53 if left paren v left bracket k right bracket less than equals twenty five right paren left brace.”

What does “graphic 53” mean in the preceding auditory cue? It turns out that when Visual Studio + JAWS reads the line that is about to be executed, it literally translates the execution arrow as “graphic 53.” If that were not confusing enough, the auditory cue would be, “graphic 409,” if the user navigated to a line that both had a breakpoint and was about to execute, and would be, “graphic 691” if the line just had a breakpoint. In contrast, when a user navigates to the line that is about to be executed in SOD, the user hears “cursor at the execution line” followed by a semantic depiction of the line itself (e.g., “if v sub k less than equals twenty five left brace”). A more complete listing of these cues can be found in Appendix B.

A third method for answering the question “How many items have been replaced after the third iteration of the loop?” is identical to the second method just described, except that the user would use the Locals Window (bottom of Figure 7.1) to inspect the contents of the array after completing three loop iterations, rather than keeping track of the number of times that the set statement on line 13 executes.

In both the SOD and Visual Studio + JAWS interfaces, the user can navigate to the Locals Window by pressing ALT + F3, at which point both SOD and Visual Studio + JAWS say “locals.” Once in the Locals Window, users of both interfaces can navigate from variable to variable using the up and down arrows. When a variable is visited, the variable name, value, and type are read in sequence. For example, suppose the user hits the down arrow to visit variable \( r \) (second from the
bottom in Locals Window of Figure 7.1). In both interfaces, the user would hear “r zero integer.”

The auditory cues for array variables differ slightly between SOD and Visual Studio + JAWS. To illustrate, suppose that the user navigates to the array variable v (top-most variable in Locals Window of Figure 7.1). In both interfaces, the user would hear the following: “v zero x zero zero one zero zero zero nine five nine zero integer” (the long sequence in the middle is the memory address of the array). In order to see the individual array values of v, the user would need to open the array by hitting the right arrow key. When the user does this in the SOD interface, the user hears “opening array v”. In contrast, hitting the right arrow key to open an array in Visual Studio + JAWS does not generate an auditory cue. Once the array is opened, the user can navigate to individual array values with the up and down arrows. The auditory cues produced for array values are identical to those produced for regular variables, with one notable exception: SOD says an array variable using “sub” language (e.g., “v sub zero”), whereas Visual Studio + JAWS says an array variable by speaking it literally (e.g., “v left bracket zero right bracket”).

7.2 Methods

7.2.1 Design

In order to evaluate the effectiveness of the SOD environment, I conducted an empirical study with the following hypothesis: In program comprehension and debugging tasks, SOD’s auditory cues will promote significantly faster and more accurate performance than the auditory cues generated by Visual Studio + JAWS; however, a visual environment will promote significantly more efficient performance than both SOD and Visual Studio + JAWS.

To test this hypothesis, I conducted a within-subjects experiment with three conditions defined by task environment:

1. SOD: An audio-only version of the environment presented in the previous section embedded with the SOD auditory cues, derived in part from work in Chapter 5;
2. JAWS: An audio-only version of the environment presented in the previous section embedded with the auditory cues produced by JAWS 9 in Visual Studio 2005 (the current state-of-the-art); and

3. Visual: The visual environment presented in the previous section with no auditory cues. This can be considered the “gold standard” that establishes an upper bound on human performance.

Comprehension and debugging outcomes were assessed according to three dependent measures: (a) accuracy on six comprehension questions; (b) ability to locate, describe, and explain how to fix two bugs; and (c) time to complete the comprehension questions and debugging tasks.

7.2.2 Participants

I recruited nineteen sighted students (13 male, 6 female; mean age 22.9) out of the Spring, 2008 offering of Computer Science 443/580, the undergraduate/graduate human computer interaction course at Washington State University. Participants were either juniors, seniors, or graduate students, and reported a mean of 5.47 years of prior programming experience. One participant, despite being told otherwise, thought aloud, which is known to be a confound for experiments involving audio interfaces (Tsujimura & Yamada, 2007). Another participant failed to complete the tasks, and was a significant outlier. These two participants were removed from the statistical analysis.

7.2.3 Materials and Tasks

All participants worked on a computer running the Windows XP operating system. While the machine was equipped with a mouse and keyboard, participants were allowed to use only the keyboard. In the Visual condition, a 19” LCD color display was set to a resolution of $1024 \times 768$. In the audio conditions, the Microsoft Mary—English (United States) voice was used as the text-to-speech speech engine.
Prior to working on the tasks in each condition, participants completed an informationally-equivalent training task that introduced them to the environment version they would be using by having them navigate the code structure, run the debugger, and use the local variable window, either with visual or auditory feedback.

Participants worked with three different program execution and debugging environments defined by condition: SOD, JAWS, and Visual. All three of these environments were identical with respect to the input commands they supported. They differed only with respect to the output they used to communicate with the user: the SOD environment used my own experimental cues, derived in part from Chapter 5; the JAWS environment used the cues generated by JAWS 9 in Visual Studio 2005; and the Visual environment presented the visual interface depicted in Figure 7.1, but had no auditory cues.

In each condition, participants worked with one of three isomorphic algorithms: Find Max (locates the largest element in an array), Replace (replaces values smaller than 25 with 0), and Count (sums the number of array values larger than 50). The task in each condition was two-fold: (a) answer six comprehension questions related to the run-time behavior of the algorithm (e.g., "How many times does the loop execute?"); and (b) locate, describe, and explain how to correct two bugs seeded in the algorithm. Participants were provided with a work booklet containing instructions for all tasks, and spaces to enter their answers to comprehension and debugging questions.

I used Morae Recorder to make lossless recordings of participants’ computer screens (which the participants themselves could not see in the SOD and JAWS conditions). An overhead camera, focused on participants’ work booklets, captured their work in a smaller inset image. These recordings allowed me to accurately gauge participants’ time on task.

7.2.4 Procedure

In order to guard against task order effects and any possible asymmetries between the tasks, I counterbalanced the task and treatment orders using a standard Latin-square design. This gave nine
possible orderings: three different task orderings crossed with three different treatment orderings. Thus, roughly two study participants performed each of the nine possible task-treatment orderings.

I ran participants through the experiment individually over the course of a ten day period. In each study session, which lasted roughly 90 minutes, participants began by filling out an informed consent form. Next, they were read a general description of what they would be doing in the study. Before performing tasks within a given treatment, participants completed a 10 minute training task for that treatment. Participants were then instructed to complete the tasks in a given treatment as quickly as possible, without sacrificing accuracy, with the stipulation that all of the tasks in a given treatment had to be completed within 20 minutes. After 20 minutes, or whenever they finished, participants moved on to the tasks in the next treatment. After finishing the tasks in all three treatments, participants were given 10 minutes to fill out an exit questionnaire.

7.3 Results

Before analyzing the data, I first verified that there were no order or task effects. A univariate analysis of variance (ANOVA) found no significant differences in (a) time for the total order, $F(7, 43) = .927, p = .495$; (b) performance in the three tasks (Find Max, Count, Replace) $F(2, 48) = .345, p = .710$; (c) nor performance with respect to task sequence, $F(2, 48) = .026, p = .974$. I conducted the same analysis on the comprehension accuracy and debugging accuracy metrics, but similarly found non-significant order and task effects. I thus concluded that my tasks were reasonably isomorphic and that the Latin square successfully removed order effects from my design. In addition, using normal probability plots, I confirmed that the accuracy and time-on-task data was normally distributed.
7.3.1 Performance Measures

Figure 7.2 and Figure 7.3 plot time on task, comprehension accuracy, and debugging accuracy by condition. Not surprisingly, these figures indicate that participants in the Visual condition promoted better performance than the other two conditions with respect to all three dependent measures. Moreover, the SOD condition appears to have promoted better performance than the JAWS condition in all three measures.

To determine if any of these differences was statistically significant, I again employed a univariate ANOVA. With respect to time on task, I found that the overall model was significant, $F(2, 48) = 15.925, p < .001$ (partial eta-squared = .399). Post hoc Tukey tests revealed that the difference in seconds between the Visual group, ($M = 452.1, SD = 216.6$), and the JAWS group, ($M = 896.2, SD = 244.3$), was significant, $p < 0.001$, and that the difference between the JAWS group and the SOD group, ($M = 601.4, SD = 238.8$), was significant, $p = 0.002$. However, the difference between the Visual group and the SOD group was not significant, $p = .160$.

With respect to comprehension accuracy, the ANOVA model was also significant, $F(2, 48) =$
4.538, \( p = .016 \) (partial eta-squared = .159). Post hoc Tukey tests revealed that the difference (number of correct comprehension questions out of 6) between the Visual group, (\( M = 4.94, \ SD = 1.09 \)), and the JAWS group, (\( M = 3.47, \ SD = 1.87 \)), was significant, \( p = .013 \). However, the difference between JAWS and SOD, (\( M = 4.47, \ SD = 1.28 \)) was not significant, \( p = .122 \), nor was the difference between the Visual and SOD groups, \( p = .615 \). With respect to debugging accuracy, the overall ANOVA model was non-significant, \( F(2,48) = 1.606, \ p = .211 \) (partial eta-squared = .063).

### 7.3.2 Survey Results

Finally, I wanted to see if participants’ subjective opinions about each environment accorded with their objective performances with each environment. Figure 7.4 plots participants’ responses to five exit questionnaire questions designed to elicit participants’ subjective opinions on each environment. All responses were on a Likert scale ranging from 1 (strongly disagree with statement) to 7 (strongly agree with statement).

I used a MANOVA to test for significant differences in participants’ questionnaire responses.
Using Wilks’ Lambda, I found the entire model was significant, $F(10, 88) = 12.801, p < .001$ (partial eta-squared = .593). According to post-hoc Tukey tests, participants found that the SOD environment’s auditory cues, as compared to the JAWS environment’s auditory cues, (a) made the task less difficult ($p < .001$), (b) made it easier to find bugs ($p < .001$), (c) made it easier to answer comprehension questions ($p < .001$), (d) were more intuitive ($p < .001$), and (e) gave more useful information ($p < .001$). While the same differences were found between the Visual and JAWS environments, differences between the Visual and SOD environments were less pronounced. In particular, while participants found that tasks with the Visual environment were easier than with SOD ($p = 0.038$), the Visual and SOD environments did not differ significantly with respect to any of the other questions.

Figure 7.4: Exit questionnaire results by condition
7.4 Discussion

The results of this experiment provide strong empirical support for my hypothesis that the SOD auditory cues promote significantly faster performance than the JAWS auditory cues. Indeed, the JAWS group took 49\% longer to complete the comprehension and debugging tasks than the SOD group, a difference that is both statistically and practically significant.

I also predicted that the Visual environment would be a “gold standard,” promoting significantly faster task performance than either of the two audio environments. I found a significant difference between the Visual and JAWS environments, with the JAWS group taking 98.3 percent longer to complete tasks; however, the 33 percent difference in task speed between the Visual and SOD environments was not found to be statistically reliable, although I suspect that it would become so, with a low effect size, if I ran more participants through the study.

With respect to comprehension and debugging task performance, I found only one statistically significant difference between the Visual and JAWS groups. Especially in the debugging task, I suspect that the lack of differences had to do with a ceiling effect: There were only two bugs to find within algorithms totaling fewer than 20 lines each, and most participants succeeded in finding the bugs. I determined in pilot testing that this ceiling effect would probably occur, but making the bugs too difficult to find could have made a ceiling effect occur for the timing metric (especially the JAWS group). As such, I decided testing more difficult bugs should be left for a future experiment.

7.4.1 Theoretical Explanation

Why was it the case that my main hypothesis, that the SOD auditory cues hold a significant advantage over the JAWS auditory cues, was substantiated? I can offer three theoretical explanations: semantic priming, auditory memory, and auditory temporal masking.

Semantic priming (Masson, 1995; Neely, 1977) is a psychological theory that suggests that a participant’s response to one stimulus can alter the speed at which the participant can mentally retrieve another stimulus. This occurs through a process called spreading activation. Why
would semantic priming afford the SOD auditory cues an advantage? Consider the auditory cues generated when the participants execute a line of code (by pressing the F9 key). In the JAWS environment, the auditory cue is “F9,” which has no meaning in the context of the executing program. In contrast, in the SOD environment, the auditory cue provides information relevant to context, such as “if true.” This information may prime the participant’s mental model of both the executing program’s behavior, as well as the participant’s temporal location within that program.

But can priming be related to something as numerical as programming? Evidence appears to suggest that the answer is yes (LeFevre et al., 1988). Consider a task that asks participants to look at the visual stimulus of $4 + 3$ on a computer screen. After a short, carefully timed, delay, participants are then flashed a second number, which is either one of the originals (either 4 or 3) or a new number (any number except 4 or 3). It turns out that if the number is considered “neutral,” a number that is not the sum of the two numbers, we can mentally determine whether the second stimuli is one of the original numbers more quickly then when the number is the sum. This special phenomenon is called the sum effect, and occurs even when the “+” is removed from the original stimuli.

This effect is not, however, universal. Recent psychological evidence from Bassok et al. (2008) suggests that the sum effect can be mitigated if certain non-similar words are attached to the numbers. Putting this result in the context of computer programming, when participants see summation operators in a program, the sum effect will naturally occur if the types of objects being summed are similar. In contrast, the sum effect is mitigated if the two summed objects were entirely dissimilar (say an overloaded $+$ operator for a CAR object and a FLOWER object in C++).

For tasks like those in this experiment, the sum effect could be playing a role, as task completion involves the human mentally processing information about the running computer program. If certain types of operators can cause delays when mentally processing mathematical operations, it is possible that, when combined, these mental gymnastics are adding to the total time to complete
the tasks. The fact that behavioral runtime cues are used in the debugger might then be an advantage. Behavioral runtime cues gives only the results of program operations, but do not require users to do mental arithmetic, like they would when only syntax cues and a watch window are available (e.g., Visual Studio 2005 + JAWS).

Priming aside, computer programmers are asked to remember a great deal of information about a piece of software they are debugging, writing, or modifying. Programmers must remember information about variables, their types, how they are being used in certain contexts, and the intent of various sections of code. Basic research on human memory suggests that when participants are asked to remember lists of words, that they have significantly better performance when the words are read out loud to them, as opposed to reading the words silently on their own (Gathercole & Conway, 1988). Conceptually, this is very similar to the current work, where one group of participants reads computer code silently, visually tracking the behavior of the code, while another hears auditory cues, essentially reading the code out loud. It is possible that the use of auditory cues benefits participants by facilitating better recall of the computer code, allowing them to complete the programming tasks quickly if the auditory cues make sense. This does not explain how SOD could perform better than Visual Studio + JAWS, but it could help explain how SOD was able to so closely match the performance of the visual interface.

In contrast to priming and memory concerns, auditory temporal masking (Zhang & Formby, 2007) is the theory that a sound stimulus can have an effect on the audibility of another stimulus either before or after the original. I suspect this phenomenon relates to the auditory cues generated when the user is navigating the code with the arrow keys. When sonifying a line of code, the JAWS environment reads every character literally, including brackets, braces, semicolons, and parentheses. It is possible that this large influx of superfluous details masks the information most relevant to the user, such as whether the line has a loop or conditional in it.

After reviewing the recordings of the experimental sessions, I find this explanation based on

---

1Although the best performance is actually gained by participants vocalizing the list.
auditory temporal masking to be especially plausible. In the JAWS sessions, participants would often have to listen to the auditory cue for each line multiple times. In contrast, in sessions with the SOD environment, which removed these extra characters and made them available only by pressing the left and right arrows, I noticed that participants rarely had to listen to the same line multiple times.

7.4.2 Threats to Validity

There were several potential threats to validity in the current experiment. First, experimental tasks were done on very small computer programs, no more than 20 or so lines of code. It is possible that the current experimental results do not generalize to large scale computer programs. It should be noted, however, that in order to test extremely large programming tasks, involving potentially thousands of lines of code, one is potentially compromising internal validity in the hopes of increasing external validity.

Second, the text-to-speech engine in use for this experiment, Microsoft Mary, is less than ideal. In the previous chapter, Microsoft Anna was used, which is substantially easier to understand. Unfortunately, it is available only for Windows Vista, and not on the Windows XP machines available in the lab where this current experiment was conducted.

In addition, about half of the participants in the current experiment were non-native English speakers; all were sighted proxies. Sighted proxies come into an experiment with little to no experience using screen readers or programming using audio based stimuli. This means that this class of users is more reflective of programmers that have recently gone blind, but is not necessarily representative of the performance of the actual non-sighted community. However, considering the current results, this is actually quite amazing. Even with no experience using a non-sighted

---

2 The careful reader might ask if non-native speakers affected the experimental results of one condition more than another. The short answer is no, they did not. While non-native speakers (M = 725.9, SD = 293.2) did perform significantly worse than native speakers (M = 564.5, SD = 278.8) for time-on-task, $F(1, 45) = 6.711, p = .013$ (partial eta-squared = .130), no significant interaction effects were found, $F(2, 45) = .684, p = .510$, implying that non-native speakers performed globally worse than native speakers, but that this effect was not significantly different across the conditions. Interaction effects were similarly non-significant for all other dependent variables.
interface, users were able to come to within a statistically negligible margin of error of sighted performance! If anything, these issues lead to an overestimation of the amount of time to complete tasks in the auditory groups (and an underestimation for comprehension and debugging).

Thus, the question that immediately comes to mind is whether real non-sighted programmers would be able to match or exceed sighted performance for the tasks in this experiment using a high quality text-to-speech engine? It is not possible to know from the current data, but the results indicate that an audio based programming interface could meet the timed performance of a sighted interface within the statistical margin of error. This is a surprising result, indeed.

7.5 Conclusion

While a large body of research has explored the design of visual representations to aid in computer programming tasks, relatively little research has explored the design of auditory representations to assist non-sighted programmers. Rather than retrofitting existing visual environments with screen readers, I have argued throughout this thesis that a superior design strategy is to build auditory representations from the ground up—that is, through a user-centered design process where the comprehension of the auditory cues is measured and the best cues are used in the environment. I have furnished empirical evidence that the auditory cues in SOD promote significantly more efficient human performance in comprehension and debugging tasks than JAWS 9 coupled with Microsoft Visual Studio 2005, the current state-of-the-art program execution and debugging environment for the non-sighted.
CHAPTER 8

CONCLUDING REMARKS

Non-sighted programmers face problems that their sighted counterparts do not. While the sighted can look at multiple parts of the screen in parallel, where each component provides unique information (e.g., a watch window and a source window), the non-sighted use serial, speech-based, auditory cues that present one type of information at a time. Further, while the sighted can look at an integrated development environment and garner meta-information from the visuals (e.g., syntax highlighting, breakpoints, details of program execution), the non-sighted need to have meta-auditory cues inserted into the audio (e.g., auditory scoping cues, capitalization).

Fundamentally, this thesis argues that we need to rethink our methodology for designing tools for the non-sighted. At the center of this problem is the need for a means of measuring the comprehension of auditory cues. By measuring comprehension we can determine what auditory cues will be easily understood (e.g., if true) and what other cues provide too little information to be useful (e.g., F9, F9, F9, F9).

My methodology for measuring comprehension is artifact encoding. This method provides a rich data set that is automatically analyzed and scored by the computer. I have demonstrated that non-sighted programming interfaces built in this way: from the ground up—by way of careful auditory cue construction and through the use of comprehension measures, are ultimately superior to retrofitted screen readers.

As this thesis has shown, the current state of the art for non-sighted interfaces is to retrofit accessibility on top of an existing interface, sometimes with little regard for usability. I argue that a paradigm shift for non-sighted interfaces is needed, where companies designing accessible products worry less about designing accessibility APIs screen readers can attach to and focus more on ensuring the audio is sensible. I have offered empirical evidence that making even minor adjustments to the auditory cues can significantly alter a human’s comprehension of what those
cues represent (see chapter 5), and changes in auditory cues can make a significant difference in the usability of a sound based interface (see chapter 7).

8.1 Contributions

There were five primary contributions in this thesis:

8.1.1 The Design Space of Auditory Cues

The literature is virtually silent on how to design effective interfaces for the non-sighted, even more so for programming environments. Yet, the only way for the non-sighted to create better tools for themselves is by programming them. I have defined three primary types of auditory cues in use for non-sighted programming environments: design, editing, and execution cues. Design cues are used to represent abstract programming structures (e.g., UML), while editing cues represent the syntax and semantics of a programming language (e.g., if left paren a equals equals b right paren left brace). Execution cues indicate the state and behavior of a running program (e.g., if true).

8.1.2 Experimental Paradigm: Artifact Encoding

Artifact encoding is an experimental paradigm for analyzing a human’s demonstrated comprehension of auditory cues. It works by asking a programmer to listen to audio and write down their interpretation. This interpretation is then coded and analyzed by a computer, creating a record of how well a programmer’s mental model of both auditory cues and the relationships between those cues match up with the designer’s intended meaning of the stimuli.

8.1.3 Empirical Study: Auditory Cues for Conveying Scope

In order to understand the scope of a construct, the non-sighted programmer needs to navigate the code looking for beginning and ending braces. To facilitate an understanding of scoping relationships without browsing, and to validate the artifact encoding paradigm, I embedded an auditory cue that represented the scope of a construct directly (e.g., 1 nested if true, 2 nested loop iteration 1). These cues were shown to effectively lower the number of scoping errors programmers made.
without negative comprehension side effects.

This artifact encoding study was also able to detect differences in the comprehension level of various classes of programmers (e.g., experienced or not experienced). While non-experienced programmers were able to use scoping cues just as effectively as experienced programmers, they tend to have lower aggregate comprehension scores and more missing data.

8.1.4 Designing an Audio Based Program Execution Environment

Putting the design space into practice, I have built a tool specifically for non-sighted computer programmers—the Sonified Omniscient Debugger (SOD). SOD is omniscient because it tracks every state change in a program as it executes, allowing forward and backward navigation. SOD is a sonified debugger because it can represent any information in the environment through auditory cues. This program execution and debugging environment was built from the ground up using basic and applied empirical studies to inform the auditory cue design and to test the environment in a realistic context.

8.1.5 Empirical Study: Screen reader vs. SOD

This thesis provides empirical evidence that programming environments built from ground up are ultimately superior to current, state-of-the-art, programming environments retrofitted with a screen reader. SOD was compared both to Visual Studio 2005 retrofitted with the JAWS 9 screen reader and a visual environment with no auditory cues. The basic results imply that with careful design, we can make programming interfaces for the non-sighted that approach the gold standard—sighted performance.

8.2 Future Research

There are several key areas in which I would like to continue this research, including automating artifact encoding studies, testing non-sighted navigation in a programming environment, creating programming languages that are easier for the non-sighted, usability testing that compares purely
visual programming environments to those with audio and visual output, and expanding SOD. These areas will be described in this section.

8.2.1 Artifact Encoding Automation

Data collected in the artifact encoding study in Chapter 5 took approximately two and a half months to grade. Each participant’s answer, for all ten tasks, needed to be coded into a string representation and entered into an Excel spreadsheet. This task is tedious and incredibly time consuming, which makes automation highly desirable.

Automating the grading process for artifact encoding is, however, somewhat difficult. An attempt at doing so would begin by creating accessible web pages, or accessible computer programs, which would allow the user to type into text boxes instead of writing their answers down on paper. This serves a dual purpose. First, since the user’s input is recorded on the computer, it can be processed electronically using a parser. Second, with computer based procedures, it is dramatically easier to include non-sighted users in an artifact encoding study, especially over the Internet.

8.2.2 Non-sighted Navigation

A second area of research I would like to explore is in the navigation of computer programs by non-sighted users. Right now, the SOD environment uses simple up and down arrows, and function keys, to allow users to navigate a single procedure. I suspect that up and down navigation in the editor will not be as efficient for the non-sighted as other techniques.

Specifically, I hypothesize that a feature to jump the cursor to certain locations, essentially intelligent and automatic bookmarks, would be a substantial improvement. For example, this intelligent bookmark interface might allow users to jump easily between methods, jump to the next or previous scope, different files, or other pieces of the program. Which program constructs would most benefit the non-sighted is not entirely clear. Further, since I have already shown evidence that, for small computer programs, non-sighted interfaces can approach the performance of sighted interfaces, is it possible that improvements in navigation can further equalize non-sighted and sighted
I suspect that good navigation commands would also require auditory cues for that navigation. It seems fair to hypothesize that these cues would function philosophically similarly to the audio glances discussed by Stevens (1996), as each auditory cue should probably summarize what parts of the code it jumped over. Spearcons may also be helpful here, as they provide extremely short auditory cues and have been shown to be effective for tasks like menu navigation (Palladino & Walker, 2007).

8.2.3 Natural Non-sighted Programming

The auditory cues in SOD currently represent the programming language C, an old language with archaic syntax and semantics. SOD then retrofits the programming language with auditory cues that represent the concepts defined by C. One pertinent question then is, would non-sighted programmers benefit from language design, built from the ground up, that is firmly grounded in the psycholinguistics (Whitney, 1998) or natural programming (Myers et al., 2004) literature?

As a trivial example of where good language design could help, consider that C still uses the characters && to indicate the concept of “and,” and || to indicate “or.” This is true despite decades of research indicating that we can mentally process words more quickly then nonsense characters that have no intrinsic meaning (Reicher, 1969). How commercial languages have managed to not evolve is mysterious; why languages like Java adopted this tyranny of atrocious syntax even more so.

My current approach for getting around syntax issues is to use semantics cues. The characters || can be easily translated into the word “or” when presented to a user. However, if the language made sense in the first place, would these semantics cues even be needed? Put another way, can we increase the effectiveness of the non-sighted further by making the syntax naturally map to meaningful word choices? Does decreasing the mismatch between semantics cues and syntax cues, by fundamentally altering the programming language, increase human performance? Do
these language alterations also benefit the sighted?

8.2.4 Visual vs. Visual + Audio

Contemporary programming environments are almost exclusively visual in nature. The audio channel is used mostly for error conditions, often indicated by a beep. The study and design of high quality auditory cues may benefit the sighted community by allowing programmers to use both visual and auditory cues when programming. Novice sighted programmers might, for example, benefit from auditory cues that reveal the semantics of the programming language. Would programmers need to remember that * is sometimes multiplication and sometimes dereference when a semantics auditory cue can inform the programmer of its usage? Would embedding auditory cues into a sighted debugger, in conjunction with the standard visual cues, further increase human performance?

8.2.5 Expanding SOD

There are several directions in which I would like to expand the Sonified Omniscient Debugger. First, I would like to integrate the custom SOD compiler and debugger into Netbeans. Netbeans 6.0 provides a highly modifiable, open source, solution for integrating custom debuggers and programming languages. By integrating SOD, Netbeans can be expanded with ground up research in non-sighted programming interface design, still allowing for backward navigation, high quality auditory cues, but maintaining the power of a large scale, commercial, programming interface.

Second, the SOD virtual machine, while sufficient for the empirical work in this thesis, is not a complete implementation of the C programming language. SOD does not allow for multiple files, does not allow for functions, and does not provide every type of syntax usually in C. Usability tests should eventually test non-sighted users’ ability to interact with these features as well.

However, rather than spending time implementing the rest of the C programming language, it seems more fruitful to design a tool for allowing me to easily alter the programming language in use in SOD. This type of system could facilitate experiments on both the auditory cues, the
programming language in use, and the mismatch between a programming language and its auditory cues.

8.3 Conclusion

This thesis promotes a bottom-up approach to accessibility. By working to create better programming environments for non-sighted programmers, we make it easier for these talented individuals to design technology for themselves (e.g. better screen readers that provide better auditory cues). However, the current state-of-the-art in programming environments for the non-sighted needs substantial improvement—a problem which this work has begun to address.
This appendix is a thorough description of the types of answers participants give in the comprehension study listed in Chapter 5. Each example consists of an answer, how that answer was coded, and the interpretation of the participant’s markings.

A.1 Simple Participant answers

In this first section, several examples are given of relatively easy to grade participant answers. These answers all follow exactly, or nearly exactly the notation system outlined in the participant’s training packet.

```plaintext
if ( T ) {
  if ( F ) {}
}
```

**Code:** ITNIFU

**Interpretation:** The participant indicated that there were two `if` statements, the second nested inside the first.

```plaintext
if ( F ) {
  if ( T ) {}
}
```

**Code:** IFNITU

**Interpretation:** Code this example similarly to the previous one. Note that participants do not write the `else` portion of false `if` statements to indicate that the statement was false. This is because, in pilot testing, participants often left the `else` out anyway, and through interviews I found
that most participants wanted to *imply* the *else*, because writing them repeatedly was tedious.

```c
while (2) {
    if (T, F) {}
}
```

**Code:** LNITU LNIFU

**Interpretation:** The participant is indicating that there was a loop that executed twice. There is one *if* statement inside the loop. On the first iteration of the loop, the *if* statement evaluated to true, while on the second it evaluated to false.

```c
while (1) {
    while (2) {
        if (T, F) {}
    }
}
```

**Code:** LN LNITU LNIFU U

**Interpretation:** Similarly to the previous example, the outer loop executed once, while the inner loop executed twice, with one *if* statement inside of it. The *if* statement was true on the first iteration of the loop and false on the second iteration of the loop.

```c
while (1)
    while (2)
        if (T, F)
```

**Code:** LN LNITU LNIFU U

**Interpretation:** It is relatively common for participants to write tabs to indicate the nesting level of constructs. Count whitespace to be indicative of nesting level.


```plaintext
if F

  while 2

  if T, F

Code: IFN LNITU LNIFU U

Interpretation: It is also relatively common for participants to either forget, or otherwise leave out, parentheses. This, I suspect, is probably due to the fact that tasks are timed and that writing parentheses is tedious.

A.2 Complex Participant Answers

In this second section, I give several examples of where a participant’s intent is not particularly clear. The interpretation and coding of these answers was derived from an analysis of participant’s shorthand, interviews after pilot studies, and answers from exit surveys taken by participants.

```plaintext
if ( T ) {
// note the closing brace is missing
  if ( T ) {
  }
}

Code: ITNITU

Interpretation: The participant forgetting the closing brace for a construct has little significance, from a comprehension standpoint. Do not code it.
```plaintext
while if ( F ) {
    if ( F, F, F ) {
    }
}
```

**Code:** LNIF IFIFIF U

**Interpretation:** The participant did not indicate the number of times the loop iterated, so assume only once. `while if ( F )` is difficult to interpret, but code it as a while with an `if` inside of it. The statement `if ( F, F, F )` is typically used to indicate each iteration of the loop. In this case, since we cannot determine the number of iterations of the loop, consider this statement to mean three serial `if` statements.

```plaintext
while {
    if ( 3xF, 3xT, 3xF, end ) {}
}
```

**Code:** LN IFITIF U

**Interpretation:** The numbers are meaningless in this context, so ignore them. The participant was likely trying to relate either nesting values, or possibly iteration values, to `if` statements. The participant was not able to, however, indicate that they grasped the meaning of the numbers. Count this passage as one iteration of a loop, since no number is given, and count the `if` statements as serial `if` statements.

```plaintext
if ( F, T, F ) {
    if ( F ) {}
}
```
**Code:** IFITIFN IF U

**Interpretation:** Typically, the comma construct is used only when inside a loop, but if, for whatever reason, a participant uses it outside a loop, count the first occurrences of the construct as serial. This particular rule is designed in this way because, during pilot testing, participants that made this mistake self reported that serial construction is usually what was meant by the notation.

```plaintext
if ( F, F, T ) {
    if ( F ) {}
else {
    if ( T ) {}
}
```

**Code:** IFIFITN IF U IN IT U

**Interpretation:** In cases like this, count the first constructs in the if as serial, and nest into the true part of the construct. Next, since it is unclear what is meant by the additional else, count it as an additional if with no truth value, which has a nested if inside of it.

```plaintext
while ( 2 ) {
    ------------------ 1
if ( T ) {}
    ------------------ 2
if ( F ) {} 
}
```

**Code:** LNITU LNIFU

**Interpretation:** Do not deduct points for the participant using the shorthand. Count the answer as the participant probably intended, if statements with different behavior on each iteration.
if ( F, T, F ) {}

**Code**: IF IT IF

**Interpretation**: If this construct exists outside the scope of a loop, interpret it as 3 serial if statements.

```
while ( 2 ) {
    -------------------------- 1
    if ( T ) {
        if ( T ) {}
    }
    -------------------------- 2
    if ( F ) {
        if ( T ) {}
    }
}
```

**Code**: LN ITNITU U LN IFNITU U

**Reasoning**: While this is incorrect notation, the participant indicated that the structure of the code, which should be accounted for by the coding. Only penalize the participant if it is not clear from the answer what the structure, and behavior, of the code was.

```
while {
    if ( T, F ) {}
}
```

**Code**: LN ITIF U
**Interpretation:** Since we do not know the number of times the participant thought the loop iterated, assume one. Do not use the shorthand as a guide to determine how many times the participant thought the loop iterated. If the participant, instead, put a “2” next to the while, this would be coded as LNITU LNIFU.

```c
while 4 {
    if ( T, F ) {}
}
```

**Code:** LNITU LNIFU LNIU LNIU

**Interpretation:** Since the participant states the loop executed four times, but only marks `if` behavior for the first two, what the behavior of the `if` statements is on the last two iterations of the loop is unclear. Count that participant knew there was an `if` statement for all four iterations of the loop, but do not give the `if` statements a truth value for the last two iterations. Do not deduct points for the participant not putting the 4 in parentheses.

```c
while ( 2 ) {
    if ( F, TTF ) {
        if ( ) {}
    }
}
```

**Code:** LNIFU LNITN ITIF UU

**Interpretation:** On the first iteration of the loop, the participant states the `if` statement was false, and as such, the `if` statement scoped within the top `if` cannot execute. On the second iteration, since the `if` was true, the nested `if` statement is executed. However, the participant wrote only one `if` statement inside of this piece of the code, making the intent unclear. Consider any non-specified arrangement of T or F combinations serial.
\[
\begin{align*}
\text{if} & \quad (T) \quad \{ \\
& \quad \} \\
\end{align*}
\]

\textbf{Code: IT IF}

\textbf{Interpretation:} While it almost looks as if the participant here wanted to indicate the second F as being nested inside the \textit{if} statement, the brackets appear to indicate that the F is outside the first \textit{if}. Second, assume that the F means an \textit{if} statement that evaluated to false.

\[
\begin{align*}
\text{if} & \quad (T) \quad \{ \\
\text{if} & \quad \text{true} \quad \text{---------------} \quad 1 \\
\text{if} & \quad \text{false} \quad \text{---------------} \quad 2 \\
& \quad \} \\
\text{else if} & \quad (T) \quad \{ \ldots \} \\
\end{align*}
\]

\textbf{Code: ITN LNITU LNIFU U}

\textbf{Interpretation:} The participant was probably trying to indicate the structure of a loop, with one \textit{if} statement in each iteration. Since the behavior with only one \textit{if} statement is easy to understand, code this as a typical loop. On the other hand, the \texttt{else if} \quad (T) makes little sense in this context, since the first \textit{if} evaluated to true, so ignore the statement.

\[
\begin{align*}
\text{IF} \quad \{ \{ \{ \{ \{ \} \} \} \} \} \} \}
\end{align*}
\]

\textbf{Code: I}

\textbf{Interpretation:} No truth about the statement is given, so do not code in a value. Do not penalize the participant for the meaningless brackets.
Code: ITN ZIFN ZITUU

**Interpretation:** The participant states that each `if` statement is part of an `else` branch in the original `if` statement. The problem, however, is that this is typically not what a participant means, as this would imply that the first `if` executed, but nothing else in their answer actually did. A literal coding of the answer would then be IT, but this is hardly fair. On the other hand, coding the answer as ITN IFN ITUU also seems unfair, as it does not take into account the fact that the participant thought the additional `if`s were related via the `else` construct. A more “fair” approach should penalize the participant while still retaining the bulk of their answer. We can do this by adding a dummy character into the coding, Z.

Add a Z into the coding for any location where the additional `else` construct is incorrectly listed. The algorithms used for grading the participant’s answer will deduct one point for each `else` incorrectly added in this way. Thus, using this technique, the participant will lose, at most, one point for every misconstrued `if` statement, as opposed to losing almost all of the points in the problem if the answer were coded as IT.

For example, given the grading described above, the participant would have globally aligned strings with the values ITN-IFN-ITUU and ITNZIFNZITUU respectively, which would give a score of 8/10, or 80%. On the other hand, if the string were coded as IT, with globally aligned strings ITNIFNITUU and ——IT–, the participant would receive a score of 2/10, or 20%, which is, in my opinion, too harsh.
```plaintext
1 if true
2 loop if false
```

**Code:** IT LNIFU

**Interpretation:** While the participant did write numbers that were supposed to indicate nesting level, they seemed to have no idea what the numbers actually meant. As such, do not nest any constructs. Count “loop if false” as one iteration of a loop and ignore the numbers.

```plaintext
if loop
   if true
```

**Code:** I LN IT U

**Interpretation:** Count this as a loop that executed once with one `if` inside of it.

```plaintext
loop 1
loop 2
```

**Code:** LNU LNU

**Interpretation:** Count this as a simple loop with 2 iterations

```plaintext
loop 1 {
   loop 2 {
   }
}
```

**Code:** LN LNULNU U

**Interpretation:** The participant may have thought that the auditory cue, “2” in “Loop Iteration 2” indicated a nesting level similarly to “2 nested.” Take the participant’s markings literally and code the second loop as being nested within the first, with however many iterations were indicated.
loop 1

loop 2

**Code:** LN LNULNU U  
**Interpretation:** Code similarly to the previous example.

```plaintext
while ( 0 )
```

**Code:** L  
**Interpretation:** The participant has indicated a loop exists in the program, but that it never executed. This is a perfectly reasonable answer and can occur in actual code! The coding of L indicates a loop that never executed, while LNU would have indicated a loop that executed once.

```plaintext
while ( 2 ) {
    if ( F, F, F, T, T ) {}
}
```

**Code:** LN IF IF IFUUU LN IT IT UU (or) LN IF IF IF LN IT IT U, depending on shorthand  
**Interpretation:** Since the participant says the loop executed twice, break the “uneven” number of if statements into two iterations. Choose whether they thought the answer was nested or serial based on the participant’s shorthand. If the participant presented a complex structure in the shorthand, do not copy it. Essentially, while the participant may have “initially heard” the shorthand and assigned it a structure, they could not demonstrate they understood how to write their answer in the form requested in the study.

```plaintext
while ( T, F )
```

**Code:** LTF  
**Interpretation:** It is possible the participant meant that two if statements executed on different
iterations of a loop. Another possibility is that the while statement first evaluated to true, but on the second iteration the loop evaluated false (in other words, it did not execute on the second iteration). Since there is no way to tell, mark only what the participant marked, that there was a loop, a T, and an F. The algorithms used in artifact encoding will be able to automatically sort out how well these markings related to an actual answer.

---

**while**

**Code**: L

**Interpretation**: The participant knew there was a loop in the code, but nothing else.

---

**while ( all )**

**Code**: LNU

**Interpretation**: Count seemingly arbitrary words as being one iteration of a loop.

---

```c
int a = 0, b = 1;
if ( a == b )
```

**Code**: IF

**Interpretation**: Execute the participant’s statements in your head and give them a value. Thus, in this case, the coding would be I then F because a is not equal to b.
APPENDIX B

SOD AUDIO INTERFACE

In this appendix, I list the auditory cues used in two various versions SOD, including my own custom auditory cues and state-of-the-art cues from Visual Studio ©2005 plus JAWS ©9. These cues were used for the experiment in Chapter 7. Keep in mind two important characteristics about the cues in this appendix. First, Visual Studio is extremely large. As such, not every visual element, nor every auditory cue, was mimicked. Second, the cues in this appendix include more features than the experiment in Chapter 5.

Besides auditory cues, the keyboard shortcuts used in Visual Studio 2005 have not been copied into SOD. By default, some of Visual Studio’s shortcuts require multiple keystrokes in sequence. Copying these verbatim served no functional purpose for my experiment. For example, in Visual Studio, the default keyboard shortcut to get to the local variable window is to first press CTRL + D, then to press L. This was changed, for all experimental groups to ALT + F3.

B.1 Visual Studio 2005 + Jaws 9

B.1.1 Characters

The following is a list of auditory cues for several ASCII characters often used in programming:

- “(” left paren
- “)” right paren
- “{” left brace
- “}” right brace
- “:” colon
- “;” semicolon
• “−” dash (minus when next to a number, as in -50)

  − \( d = d - d + 2 \); would be “d equals d dash d plus 2 semicolon”

  − \( d = d - 50 + 9 \); would be “d equals d minus 50 plus 9 semicolon”

  − \( d = d - 50 + 9 \); would be “d equals d dash 50 plus 9 semicolon”

• “+” plus

• “[” left bracket

• “]” right bracket

• “<” less than

• “>” greater than

• “∗” star

B.1.2 Arrow Keys

Example auditory cues for the line: \( \text{if}(d < 5)\ \{ \)

• Up Arrow: Speak the line at the cursor: “if left paren d less than 5 right paren left brace.”

• Down Arrow: Speak the line at the cursor. Auditory cue is the same as above.

• Right Arrow: Speak the character at the cursor.

• Left Arrow: Speak the character at the cursor.

B.1.3 Special Arrow Key Functions

• Cursor at top line in file: No auditory cue

• Cursor at bottom line in file: No auditory cue
• Cursor at a line that contains only white space: “Blank”

• Cursor at the line that is highlighted in the debugger: “Graphic 53”

• Cursor at the line that is highlighted in the debugger, with a breakpoint at that line: “Graphic 409”

• Cursor at a line with a breakpoint: “Graphic 691”

B.1.4 Local variable window

• Up Arrow: Speak the variable name, value, and type.

• Down Arrow: Speak the variable name, value, and type.

• Right Arrow: No auditory cue

• Left Arrow: No auditory cue

• Variable: Example, “d -1000 integer”

• Array: Example, “m 0x0012ff30 integer”

• Focus at top of variable list: No auditory cue

• Focus at bottom of variable list: No auditory cue

B.1.5 Navigation

• Source window: Key press, file name, “edit pressed unavailable type in text” Example: “F7, Find Max dot c p p edit pressed unavailable type in text”

• Local variable window: “Locals” Jaws behavior here is not entirely consistent. Sometimes Jaws will say the word, “Locals” whereas other times it will say the words, “star locals.” Other times still it will say, “Locals” and then the line in the local variable window that currently has the focus.
• Compiler error window: “Error list list box” number of items “items, to move to an item press the arrow keys.” Why JAWS says, “list list” is not clear.

B.2 Sonified Omniscient Debugger

B.2.1 Characters

The following is a list of auditory cues for several ASCII characters often used in programming:

• (“” left paren
• “)” right paren
• “{” left brace
• “}” right brace
• “:” colon
• “;” semicolon
• “−” minus
• “+” plus
• “[” left bracket
• “]” right bracket
• “<” less than
• “>” greater than
• “∗” star
B.2.2 Arrow Keys

Example auditory cues for the line: \( \text{if ( } d < 5 \text{ ) } \{ \}

- Up Arrow: Speak the line at the cursor: “if d less than 5 left brace.”
- Down Arrow: Speak the line at the cursor. Auditory cue is the same as above.
- Right Arrow: Speak the character at the cursor.
- Left Arrow: Speak the character at the cursor.

B.2.3 Special Arrow Key Functions

- Cursor at top line in file: “The top line in the file is”
- Cursor at bottom line in file: “The bottom line in the file is”
- Cursor at a line that contains only white space: “Blank”
- Cursor at the line that is highlighted in the debugger: “Cursor at the execution line”
- Cursor at the line that is highlighted in the debugger, with a breakpoint at that line: “Breakpoint, Cursor at the execution line”
- Cursor at a line with a breakpoint: “Breakpoint”

B.2.4 Local variable window

- Up Arrow: Speak the variable name, value, and type.
- Down Arrow: Speak the variable name, value, and type.
- Right Arrow: If the variable is a container, like an array, the cue is, “Opening variable” variable name
• Left Arrow: If the variable is a container, like an array, the cue is, “Closing variable” variable name

• Variable: Example, “d -1000 integer”

• Array: Example, “m 0x0012ff30 integer”

• Focus at top of variable list: “The top local variable is”

• Focus at bottom of variable list: “The bottom local variable is”

B.2.5 Navigation

• Source window: file name, “entered the text editor” Example: “default dot c p p entered the text editor”

• Local variable window: “Locals”

• Compiler error window: “Compiler Messages”
BIBLIOGRAPHY


