PROGRAMMING BY VOICE:
A HANDS-FREE APPROACH FOR
MOTORICALLY CHALLENGED CHILDREN

by

AMBER WAGNER

DR. JEFF GRAY, COMMITTEE CHAIR

MARCUS BROWN
NICHOLAS KRAFT
RANDY SMITH
SUSAN VRBSKY

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2015
ABSTRACT

Computer Science (CS) educators frequently develop new methodologies, languages, and programming environments to teach novice programmers the fundamental concepts of CS. A recent trend has focused on new environments that reduce the initial challenges associated with the heavy syntax focus of textual programming languages. There are numerous Initial Programming Environments (IPEs) available that have been created for student use that in some cases have fostered self-discovery and inquiry-based exploration. In this dissertation, three IPEs are discussed: Scratch (2015), Lego Mindstorms (2015), and Blockly (2015).

Although the block-based nature of IPEs can be helpful for learning concepts in CS, a small group of students (approximately 5%) is being left out from learning experiences and engagement in CS due to block-based environments’ dependence on the Windows Icon Mouse Pointer (WIMP) metaphor. Block-based environments often require the use of both a mouse and keyboard, which motorically challenged users often are unable to operate. Based on research performed and presented in this dissertation, a Vocal User Interface (VUI) is a viable solution that offers a “Programming by Voice” (PBV) capability (i.e., a capability to describe a program without using a keyboard or mouse).

However, adapting legacy applications can be time consuming, particularly, if multiple applications (such as the three IPEs previously mentioned) require specialized VUIs. Each environment has its own visual layout and its own commands; therefore, each application requires a different VUI. In order to create a more generic solution, a Domain-Specific Language
(DSL) can be applied to create a semi-automated process allowing a level of abstraction that captures the specific needs of each IPE. From the specification of each IPE, a customized VUI can be generated that integrates with the legacy application in a non-invasive manner.

The nine chapters included in this dissertation were motivated by the following four research questions:

1. How can we improve initial programming instruction?
2. Can all children participate in programming instruction?
3. How do we implement PBV to allow children to take advantage of creative, block-based programming environments?
4. What are some potential ideas that can assist in generalizing the process of voice enabling IPEs?
DEDICATION

To my loving husband, Ray, and my adorable daughter, Izzy, thank you for your support and faith.

To my mentor, Isabel, thank you for your guidance.

Philippians 4:13
# LIST OF ABBREVIATIONS AND SYMBOLS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>AP</td>
<td>Advanced Placement</td>
</tr>
<tr>
<td>CDC</td>
<td>Center for Disease Control</td>
</tr>
<tr>
<td>CMU</td>
<td>Carnegie Mellon University</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Science</td>
</tr>
<tr>
<td>CSCP</td>
<td>Computer Science Collaboration Project</td>
</tr>
<tr>
<td>CSTA</td>
<td>Computer Science Teachers’ Association</td>
</tr>
<tr>
<td>DART</td>
<td>Daily Automated Regression Tester</td>
</tr>
<tr>
<td>DO-IT</td>
<td>Disabilities, Opportunities, Internetworking and Technology</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DTT</td>
<td>Dialog Traversal Testing</td>
</tr>
<tr>
<td>EATA</td>
<td>Educational and Assistive Technology Application</td>
</tr>
<tr>
<td>EFG</td>
<td>Event-Flow Graph</td>
</tr>
<tr>
<td>EIC</td>
<td>Event Interaction Coverage</td>
</tr>
<tr>
<td>EIG</td>
<td>Event-Interaction Graph</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human Computer Interaction</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>IPA</td>
<td>International Phonetic Alphabet</td>
</tr>
<tr>
<td>IPE</td>
<td>Initial Programming Environment</td>
</tr>
<tr>
<td>IVR</td>
<td>Interactive Voice Response</td>
</tr>
<tr>
<td>JSAPI</td>
<td>Java Speech Application Program Interface</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Location and Context Aware Learning</td>
</tr>
<tr>
<td>MEEC</td>
<td>Minimal Effective Event Context</td>
</tr>
<tr>
<td>MIT</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>MS</td>
<td>Microsoft</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>NIH</td>
<td>National Institute of Health</td>
</tr>
<tr>
<td>OCR</td>
<td>Optical Character Recognition</td>
</tr>
<tr>
<td>PBV</td>
<td>Programming by Voice</td>
</tr>
<tr>
<td>RAPTOR</td>
<td>Rapid Algorithmic Prototyping Tool for Ordered Reasoning</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SPEED</td>
<td>for SPEech EDitor</td>
</tr>
<tr>
<td>STEM</td>
<td>Science, Technology, Engineering, and Mathematics</td>
</tr>
<tr>
<td>TACCESS</td>
<td>ACM Transactions on Accessible Computing</td>
</tr>
<tr>
<td>TAIT</td>
<td>Train and Assess Information Technology</td>
</tr>
<tr>
<td>TCI</td>
<td>Translator-Computer Interaction</td>
</tr>
<tr>
<td>UA</td>
<td>University of Alabama</td>
</tr>
<tr>
<td>UAT</td>
<td>User Acceptance Testing</td>
</tr>
<tr>
<td>UCP</td>
<td>United Cerebral Palsy of Greater Birmingham</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>USF</td>
<td>University of San Francisco</td>
</tr>
<tr>
<td>VASDE</td>
<td>Voice-Activated Syntax-Directed Editor</td>
</tr>
<tr>
<td>VRT</td>
<td>VUI Review Testing</td>
</tr>
<tr>
<td>VUI</td>
<td>Vocal User Interface</td>
</tr>
<tr>
<td>WIMP</td>
<td>Windows Icon Mouse Pointer</td>
</tr>
<tr>
<td>WOZ</td>
<td>Wizard of Oz</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First, I would like to acknowledge my husband, Raymond Wagner, without whom I would never have had the opportunity to complete this degree. His support throughout this process was incredible, and I am forever grateful for his faith in me. Dinner conversations were always interesting and extremely helpful to my work, and I am so thankful that he never hides his infinite love for me.

I would like to thank Isabel Rosenblitt who inspired me from my CS beginnings in undergrad. She advised me in ways no one else could and was always willing to serve as a sounding board for my work. There is not a day that goes by that I do not think of her and miss her. She was a pioneer in the CS field and paved the way for young women like me to study CS and work in the field.

Without the help of my parents, especially during this past year when my daughter was born, I never would have had the ability to finish. Their never ending support is always appreciated, and I am thankful they raised me to be the person I am today.

I would like to thank Dr. Jeff Gray for all of his help and guidance throughout this process. It is with his encouragement and support that I am able to produce this research. Dr. Gray has been an incredible mentor throughout this entire process from advising me on courses to guiding my research, and the opportunities he provided me with over these past five years will never be forgotten.
I would also like to thank my Committee members: Dr. Susan Vrbsky, Dr. Marcus Brown, Dr. Randy Smith, and Dr. Nicholas Kraft, who provided me with assistance during my studies from providing research material to lending an open ear during stressful times.

It was an absolute pleasure to work with Dr. Gary Edwards at United Cerebral Palsy of Birmingham. He and his staff express their passion for working with children and adults with disabilities daily. It is by their example that this same passion has blossomed within me, and I hope to keep persons with disabilities at the center of my future research endeavors.

This project was supported by NSF grant IIS-1117940, with previous support from a Google Research Award.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>iv</td>
</tr>
<tr>
<td>LISTS OF ABBREVIATIONS</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xv</td>
</tr>
<tr>
<td>LIST OF LISTINGS</td>
<td>xviii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. CS Education on the Rise</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Why Initial Programming Environments are Used Heavily</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Addressing Accessibility Needs of IPEs</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Dissertation Roadmap</td>
<td>16</td>
</tr>
<tr>
<td>2. TECHNOLOGY EDUCATION AND SPECIAL NEEDS</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Computers and People with Disabilities: Then and Now</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Technology in the Classroom</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Technology in the Special Needs Classroom</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Getting Involved</td>
<td>27</td>
</tr>
<tr>
<td>2.5 Conclusion</td>
<td>31</td>
</tr>
</tbody>
</table>
3. BACKGROUND RESEARCH ................................................................. 33
   3.1 Ability-Based Design ................................................................. 34
   3.2 Programming by Voice ............................................................. 38
   3.3 Vocal Limitations ................................................................. 43
   3.4 Additional Input Modalities .................................................... 44
   3.5 Conclusion ........................................................................... 46

4. MYNA: HANDS-FREE PROGRAMMING ....................................... 47
   4.1 Overview of Early Myna .......................................................... 48
   4.2 Learning from Scratch .......................................................... 57
   4.3 GUI Mapping Challenges ....................................................... 62
   4.4 Myna Requirements, Limitations, and User Assumptions ........... 70
   4.5 Conclusion ........................................................................... 71

5. USER EVALUATIONS OF MYNA .................................................... 72
   5.1 UCP Client Review Evaluation and Discussion ....................... 72
   5.2 Pilot Study Evaluation and Discussion ...................................... 74
   5.3 Target User Evaluation and Discussion ................................... 80
   5.4 User Testing Data Discussion .................................................. 83
   5.5 Ability-Based Design Evaluation ............................................. 86
   5.6 Lessons Learned from Evaluations ......................................... 89

6. ADAPTING MYNA BASED ON USER EVALUATIONS .................. 91
   6.1 Editing Parameters ................................................................. 91
   6.2 Grammar Customization ....................................................... 92
   6.3 Delete Program and Start Over .............................................. 94
LIST OF TABLES

1.1 List of 4 major diagnoses impacting users’ abilities ........................................... 15

2.1 University of Alabama summer camp schedule .................................................... 22

4.1 Summary of teacher data collected ....................................................................... 59

4.2 Summary of student data collected ....................................................................... 60

5.1 Summary of observation data ................................................................................ 77

5.2 Average number of speech recognition errors per person .................................... 79

5.3 Error counts for Participant One ............................................................................ 82

5.4 Error counts for Participant Two ............................................................................ 82

7.1 A comparison of the times taken to create property files ....................................... 111

7.2 Description of each lexicon within MynaScript ..................................................... 114

7.3 Lines of code (loc) needed to write “delete” and “drag and drop” ....................... 115
LIST OF FIGURES

1.1 AP Computer Science Participation (2002-2014) ............................................. 2
1.2 Scratch users worldwide .................................................................................. 7
1.3 MIT's Scratch v1.4 interface ............................................................................ 7
1.4 Lego Mindstorms interface ............................................................................. 9
1.5 Blockly example on Google website ................................................................. 10
1.6 Pixly example .................................................................................................. 10
1.7 Spherly example .............................................................................................. 11
1.8 Dissertation roadmap ...................................................................................... 16
2.1 A participant solving an App Inventor program ............................................... 29
2.2 CS Unplugged activity ...................................................................................... 30
3.1 Seven principles of ability-based design .......................................................... 35
3.2 Vocal Joystick vowel mapping ......................................................................... 42
4.1 Myna architecture .......................................................................................... 49
4.2 Scratch with the three types of components identified .................................... 51
4.3 Myna Workflow – mapping of voice commands to actions in Myna ............. 55
4.4 Myna and use of transparent frames ................................................................ 56
4.5 Passing parameters .......................................................................................... 63
4.6 Editing a parameter with Myna ....................................................................... 64
4.7 Deleting a middle block in Scratch .................................................................. 66
4.8 Deleting a middle block with Myna ................................................................. 68
4.9 Variables menu ........................................................................................................69
5.1 Pilot study design ....................................................................................................74
5.2 Time to complete using mouse/keyboard vs. Myna ...........................................78
5.3 Sum of errors per program ....................................................................................78
5.4 Sum of speech recognition errors per program by speaker .................................80
5.5 Survey results per design guideline ....................................................................88
6.1 “Edit two at five” ....................................................................................................92
6.2 Dropdown list for customized grammar wizard ..................................................93
6.3 Example use of customized grammar wizard .....................................................93
6.4 Warning dialog displayed when user issues “clear” command .........................95
6.5 Speech recognition display window ....................................................................97
6.6 Location of speech recognition display window ...............................................98
6.7 Speech recognition window when misrecognition occurs .................................99
6.8 Help message displayed when first starting Myna ............................................101
6.9 Help message displayed after invoking “drag” ..................................................101
6.10 Help message displayed after invoking “edit” ..................................................101
7.1 The generalization of Myna for customized applications ............................103
7.2 Initial MynaMapper screen when starting a New program ..............................108
7.3 Collect Information menu ..................................................................................109
7.4 MynaMapper data collection screen for a movable component ......................109
7.5 Scratch overlayed with MynaMapper example ...............................................110
8.1 EIG for Open menu item based on grammar example ........................................125
8.2 Software Development Life Cycle for VUI with testing components .......... 126

8.3 “Angry Birds” program from Code.org ................................................................... 127
LIST OF LISTINGS

4.1 Property file for the “add” block ................................................................. 50
4.2 “Drag and drop” code ................................................................................. 53
4.3 Algorithm identifying which parameter to edit ........................................ 65
4.4 Algorithm to change the value of a parameter ........................................ 65
4.5 Delete algorithm ......................................................................................... 67

7.1 MynaScript DSL ......................................................................................... 113
7.2 MynaScript of “delete” and “dragAndDrop” methods for Scratch .......... 114
7.3 MynaScript of “delete” and “dragAndDrop” methods for Mindstorms .... 115
7.4 Generated code from MynaScript .............................................................. 116
CHAPTER 1
INTRODUCTION

Currently, there are several nationwide initiatives (e.g., CS10K\textsuperscript{1}, CS4HS\textsuperscript{2}, Code.org\textsuperscript{3}) whose focus is to increase Computer Science (CS) education. A particular strategy is to begin teaching Initial Programming Environments (IPEs) (e.g., Scratch (2015), Lego Mindstorms (2015), App Inventor (2015)) in grades 4-12. Unfortunately, these IPEs exploit Graphical User Interfaces (GUIs) requiring a mouse and keyboard, which does not allow for an inclusive classroom for any student with a motor impairment. The primary focus of this research is to provide a method for motorically challenged children (i.e., children with limited limb mobility) to utilize IPEs in the hope of providing these children with an inclusive classroom experience, in addition to skills from which they might benefit in future employment opportunities.

1.1 CS Education on the Rise

According to the College Board, 2,342,528 students took 4,176,200 exams from the College Board’s Advanced Placement (AP) program in 2014; however, only 39,278 (1.7\%) students participated in AP Computer Science (College Board, 2015). While this participation rate is low, it has increased by 50\% in the past two years. Figure 1.1 references the trajectory of student participation in the AP Computer Science program over the past decade and demonstrates an increase in recent years, which is likely due to a change in the pedagogy in

\textsuperscript{1} https://cs10kcommunity.org
\textsuperscript{2} http://www.cs4hs.com
\textsuperscript{3} http://code.org
addition to support from nationwide initiatives (e.g., CS10K, CS4HS, Code.org). In fact, AP CS A had the fastest growing enrollment in 2013-2014 among all College Board AP exams.

The Computer Science Teachers Association (CSTA) conducted two surveys: one in 2005 and an identical one in 2007 regarding High School Computer Science. In the analysis of the two surveys, Gal-Ezer and Stephenson (2009) attribute the decrease in AP CS to three primary factors:

1. Rapidly changing technology;
2. Lack of student interest; and
3. Lack of staff support.

More recent CSTA research (Gal-Ezer & Stephenson, 2010) echoes rapidly changing technology as a problem for developing CS teacher certification problems. Although teachers cannot control the speed at which technology changes nor the amount of staff support available, teachers can provide a pedagogically sound yet interesting curriculum to motivate students rather than discourage them. The College Board has been working on a new AP exam (CS Principles),

---

4 The College Board discontinued the AP CS AB exam in 2009 due to low participation.
which will be offered to students as an introduction to CS. This course will most likely be taught using an IPE such as Scratch or App Inventor because the current generation of students will not be impressed with the traditional “Hello World” program in a textual language (Kelleher & Pausch, 2005; Wolber, 2011) and pedagogical instruction techniques used with previous generations of students. This current generation of students has been playing video games with advanced graphics since they learned to walk, and the notion of internet accessibility is often viewed as a common utility. To increase the interest and attention of students, exciting new programs and technologies are being developed (e.g., Scratch, App Inventor, Lego Mindstorms). Additionally, student engagement is often more successful when the context is driven by current topics that are of direct interest to students. For example, most teenagers are frequent users of smartphones, which provide a unique context for engagement. The adoption rate of cellphones among students age 13-17 in the US continues to grow, and it has been estimated that 73% of teens own a smartphones (Pew Research Center, 2015). Based on the US census results (U.S. Census Bureau, 2014), this suggests that there are roughly 15.4M middle and high school students with smartphones. Educators can take advantage of these devices as a springboard for motivating topics involving CS (College Board, 2015; Fenwick et al., 2011; Loveland, 2011). Mobile application development is not the only way to reach students, but based on the statistics, it is clearly an excellent starting point.

This dissertation describes work to enable students with physical disabilities to participate in classrooms where more graphical IPEs are being used to teach introductory CS topics. The graphical IPEs use a block-based structure where users drag and drop blocks of code onto an editor. The next section describes block-based IPEs in more detail.
1.2 Why Initial Programming Environments are Used Heavily

Based on the statistics for AP CS participation presented in the previous section, methodologies and tools that introduce new students to CS and programming must be improved. Papert (1980) suggests that programming languages should satisfy three conditions: low-floor (simple to start using), high-ceiling (ability to increase difficulty and complexity over time), and wide-walls (ability to create various types of projects). The current approach instructors use to meet these suggested requirements is to utilize an environment with a block-based interface that eliminates the concern of syntax (e.g., Scratch, Alice (Alice, 2015), Lego Mindstorms, and App Inventor) because their focus is on education and pedagogical concerns, rather than the traditional focus of a development environment for a programmer. Based on observations and empirical studies (Malan & Leitner, 2007), the use of IPEs offers numerous benefits, such as: removal of the syntactic details of a language, the fostering of creativity, increasing engagement and interest, and the ability to initiate problem solving immediately (Carlisle et al., 2005).

Since 1963 (Kelleher & Pausch, 2005), researchers at universities have been developing IPEs to introduce programming concepts and problem solving with languages aimed at simplifying syntax and expanding the groundbreaking simplicity introduced by Papert in 1967 with Logo (Kelleher & Pausch, 2005; Papert, 1980). Some environments (e.g., BASIC, Turing, and GNOME) were designed for college-level beginning programmers while others (e.g., Play, LogoBlocks, and Alice) were built for a younger target audience, such as elementary school children. The environments (e.g., Karel, GRAIL, and LegoSheets) reviewed by Kelleher and Pausch (2005) “tried to make programming accessible in three main ways, namely, by simplifying the mechanics of programming, by providing support for learners and by providing students with motivation to learn to program.” To address the concern over the challenges of
learning the syntax of a programming language, many of the current IPEs have a form of “drag and drop” programming whereby a student connects program constructs together in a manner that forces correct syntax (but may still produce logic errors).

The following three learning environments are explained in more detail in the following subsections due to their use in this dissertation: Scratch (2015), Lego Mindstorms (2015), and Blockly (2015).

1.2.1 Scratch

MIT created Bongo in 1997, which allowed users to create their own video games and then share them with friends via the web (Kelleher & Pausch, 2005). The Massachusetts Institute of Technology (MIT) followed this same community-driven sharing model when creating Scratch, which was introduced in 2004 as an IPE that is simplified enough to teach to third graders, yet complex enough to teach to college freshmen or non-CS majors (Resnick et al., 2009). The University of California, Berkeley has adapted MIT’s version of Scratch to a more advanced version called SNAP! (2015), which Berkeley teaches to freshmen and non-CS majors. Resnick, the creator of Scratch, studied how children in kindergarten learn, and he found the following cyclical pattern: imagine, create, play, share, reflect (Resnick, 2007). Thus, the introductory CS applications developed by the MIT Media Lab provide young learners with the opportunity to perform this cycle (e.g., Pico Crickets, Scratch). Scratch allows this pattern of thinking by following three critical design principles (Resnick et al., 2009): be tinkerable, be meaningful, and provide a social environment. Scratch should be tinkerable in allowing users to create something quickly, similar to rapid prototyping (Resnick et al., 2009). Scratch is meaningful in that users can choose from a variety of programs (e.g., game, animation) and upload custom artwork to personalize their programs, thereby making the program more
individual overall. Scratch developers included a forum for the Scratch community that provides users with a place to share their programs and comment on other users’ creations. By providing a creative, tinkerable, meaningful, and social environment, Scratch appeals to all ages and levels, not just to children (Resnick et al., 2009).

The creators of Scratch at MIT intended for Scratch to be a “networked, media-rich programming environment designed to enhance the development of technological fluency at after-school centers in economically-disadvantaged communities” (Maloney et al., 2004). The Scratch community has emerged into a social network where 799,212 design studios have been created displaying 8,672,833 projects for all 5,935,023 (up from 1,519,271 in 2012) registered members to view (Scratch, 2015; Scratch Statistics, 2015). Kelleher and Pausch (2005) observed that social environments such as networked applications provide more motivation for students to learn how to program, and Scratch has provided further evidence to support this observation (Resnick et al., 2009). Scratch is visited daily from 150 countries in 40 languages (see Figure 1.2) (Scratch, 2015; Scratch Statistics, 2015). As an IPE, Scratch is reaching millions of students and educators daily. The projects that are shared within the community serve to inspire other members in numerous ways (e.g., students share their culture, political and religious views, students collaborate to create scripts other students can utilize, and students learn how to critique one another) (Scratch, 2015). Scratch is more than just a programming environment; it is a socially-driven, creative tool that children can use to express themselves and explore their own individuality.

Scratch is heavily utilized throughout the world because of its ease of use and general availability (Scratch is freely available). Figure 1.3 is an image of the Scratch UI. All of the commands are “drag and drop” allowing the programmer to focus on logic and problem solving

5 https://scratch.mit.edu
rather than syntax. Also, the programmer can develop his or her own images and import sounds, thereby fostering a creative environment that makes the underlying difficulty of the program transparent to the student.

In 2013, MIT released Scratch v2.0. While very similar to the previous version (1.4), Scratch 2.0 is entirely web-based (like SNAP!). Additionally, the creators of Scratch 2.0 added the functionality of creating custom blocks (Scratch, 2015).

![Map of Scratch users worldwide](image1.jpg)

**Figure 1.2.** Scratch users worldwide (Scratch, 2015).

![MIT's Scratch v1.4 interface](image2.jpg)

**Figure 1.3.** MIT's Scratch v1.4 interface (Scratch, 2015).
1.2.2 Lego Mindstorms Software

The University of Colorado created LegoSheets in 1995 to work with the MIT Programmable Brick (“intelligent, built-in computer-controlled LEGO brick that lets a MINDSTORMS robot come alive and perform different operations” (Ranganathan et al., 2008)). LegoSheets was primarily graphical and allowed users to progress slowly from controlling the basic motor functionality of a robot to adding conditional statements in a program. In 1996, the MIT Media Lab developed LogoBlocks, which was “a graphical programming language designed for the Programmable Brick” (Kelleher & Pausch, 2005). With LogoBlocks, users could “drag and drop” code blocks to manipulate the brick and also allowed the user to learn about the creation and use of parameterized procedures to support generic reusable functionalities.

The successor of the programmable brick is the commercially available Lego Mindstorms environment developed by the MIT Media Lab in 1998. Since then, the programming environment has improved along with the hardware (e.g., the initial RCX Mindstorms robot used a very fragile infrared process for downloading programs, but the newer EV3 platform supports Bluetooth and USB connections). A special programming template has been developed for National Instruments’ LabVIEW (National Instruments, 2015), which provides a graphical programming environment that is used by many K-12 schools, including many in the state of Alabama (e.g., Alabama School of Fine Arts, Northridge High School, Tuscaloosa Magnet, Rock Quarry Middle School). The software comes with 46 tutorials to assist educators and/or students in learning how to program and command the EV3 robot. The tutorials demonstrate how to build the robot (this is helpful if specific sensors are needed) and how to build the blocks for the code (Lego Mindstorms Software, 2015). Lego has created a social media forum for users where
members can post their projects and share ideas in the Mindstorms Gallery\(^6\), similar to Scratch. Figure 1.4 is a screenshot of the Mindstorms interface (Lego Mindstorms Software, 2015), which is similar to MIT’s original design. There is a block palette on the left, the program editor in the center, and the tutorials are on the right.

In 2014, a new version of the Lego Mindstorms interface was released in coordination with the new Lego Mindstorms EV3 robot. Both are very similar to their predecessors, but regarding the software, a few simplifications were made within the blocks (i.e., there are fewer parameters due to a re-categorization of the blocks) (Lego Mindstorms Software, 2015).

1.2.3 Blockly

Google’s Blockly (2015) enables developers to take advantage of a library to build visual programming editors. Many existing IPEs were built using Blockly, which was influenced by App Inventor (a visual programming language that allows users to write apps using a block-oriented “drag and drop” environment to create an app’s user interface and to specify its behavior and functionality) (App Inventor, 2015). A few other environments built using Blockly

---

\(^6\) http://www.lego.com/en-us/mindstorms/gallery

Work has been done at the University of Alabama using Blockly to build Pixly, a graphical IPE for image manipulation (based on Georgia Tech’s media computation work (Media Computation, 2015)) (Trower & Gray, 2015; Pixly, 2015), and Spherly, a graphical IPE for programming the Sphero robot, which is a robotic ball being used to increase enthusiasm for CS in middle and high schools (Trower & Gray, 2015; Spherly, 2015). Pixly (2015) allows a user to programmatically alter the pixels of an image (see Figure 1.6). Spherly (2015) allows a user to write programs to control the Sphero robot (see Figure 1.7). As both of these images

Figure 1.5. Blockly example on Google website (Blockly, 2015).

Figure 1.6. Pixly example (Pixly, 2015).
illustrate, Pixly and Spherly are simple, block-based languages, and they look similar to one another due to being developed using the same Blockly library.

1.2.4 Observations of Common and Distinguishing Characteristics among Modern IPEs

Scratch, Lego Mindstorms, and Blockly have a common ancestor: LogoBlocks (Begel, 1996). LabVIEW is a direct descendent of LogoBlocks (Kelleher & Pausch, 2005), Lego blocks were the inspiration for Scratch (Resnick et al., 2009), and Blockly followed the trend set ultimately by LogoBlocks in 1996. Scratch was intended for students ages eight to 16 (Resnick et al., 2009); however, Malan and Leitner (2007) found success using it in introductory CS courses at the college-level. Similarly, Lego Mindstorms was designed for K-12 classrooms, yet Ranganathan, Schultz, and Mardani (2008) found success using it in a freshman Electrical Engineering course. IPEs developed using Blockly have a varied age range, but they are similar to those of Scratch and Mindstorms (e.g., Code.org is used in K12 classrooms (Code.org, 2015), App Inventor (2015) is used in K12 and University classrooms).

While Scratch, Mindstorms, and IPEs developed using Blockly have many commonalities (e.g., similar age range, graphical blocks that snap together in a program editor, and a command palette), there are differences among them. Mindstorms and Spherly (a Blockly-based IPE) have a physical component that allows students to observe their programs run on a
device other than the computer. Because of the associated physical device, Mindstorms has a duality in that the user must ensure the robot is built appropriately. Although Scratch is intended to run on the computer only and is not associated with a physical device, it does have a similar duality as users must design the stage (or background) of their world prior to building the program. Depending on the IPE developed using Blockly, it may have this physical duality; however, Spherly, Pixly, and Code.org do not require the development of an additional interface.

An obvious commonality among all of these graphical programming environments is the dependence on the Windows Icon Mouse Pointer (WIMP) metaphor, which “provides ease of use but assumes dexterity of human hands to use a mouse and keyboard” (Wagner et al., 2012). Clearly, this dependence results in a failure to achieve universal usability (Wobbrock et al., 2011) and “[address] the needs of all users” (Shneiderman & Plaisant, 2010). Based on the ACM code of ethics (ACM, 2015), it is important that these accessibility issues be addressed.

1.3 Addressing Accessibility Needs of IPEs

As previously mentioned, many of the current IPEs have a form of “drag and drop” programming. However, because this new breed of IPEs requires motor functionality to operate a mouse and keyboard, those students who are motorically challenged are unable to participate in these new learning experiences. The ACM code of ethics states, “[i]n a fair society, all individuals would have equal opportunity to participate in, or benefit from, the use of computer resources regardless of race, sex, religion, age, disability, national origin or other such similar factors” (ACM, 2015). By not providing an alternative means of access, children with disabilities are being denied learning opportunities that may allow them to explore career paths in computing. Moreover, on June 29, 2010, the Department of Education distributed a letter to colleges and universities across the country requiring that any technology used in a classroom be
fully accessible to all students. Although the letter focuses on visually impaired students, not providing alternative accessibility options to any disabled student violates the Americans with Disabilities Act of 1990 and the Rehabilitation Act of 1973.

Understanding how users with motor impairments currently use a computer and what issues they have with a standard computer is imperative to creating a better solution to “fit the abilities” (Wobbrock et al., 2011) of these users. Harada et al. (2009) performed a study in which five of the participants suffered from various degrees of motor impairment. A description of how three of those participants were able to interact with a computer follows. Participant number one was 52 years old and had been afflicted with multiple sclerosis for seven years at the time of the study. Although he can type, he uses Dragon Naturally Speaking instead because he easily tires from the arthritic pain he feels in his hands. Participant number three (age 20) has suffered from muscular dystrophy since birth. Muscular dystrophy causes “progressive weakness and degeneration of the skeletal muscles that control movement” (NIH 2012); thus, this participant has a small range of mobility and must be in a wheelchair. Because her mobility is limited to the point that she cannot turn her hands so that her palms are face down, she has taught herself how to type with the backs of her hands. Participant number four (age 30) was diagnosed with cerebral palsy at birth. Due to her condition, she has violent spasms frequently and weakened muscles. In addition to cerebral palsy, she suffers from fibromyalgia, a “chronic condition” (Harada et al. 2009) causing the nerves to misfire constantly. Her input device of choice is a touchpad on her laptop because the traditional keyboard and mouse are tiring for her.

Changing input devices from a traditional device to a new input modality (e.g., voice, touchpad, eye tracking) is a very difficult process; both the user and the application must learn or be trained, respectively. With voice as an input modality, the user needs to learn the grammar,
and the application needs to learn the user’s accent/articulation (Begel, 2005; Dai et al., 2004; Désilets et al., 2006; Harada et al., 2009; Shaik et al., 2003; Wobbrock et al., 2011). Because the traditional input devices for a computer cause extreme fatigue for individuals suffering from these conditions and they truly are not designed with their specific needs in mind, these individuals are willing to devote the extra time it takes to learn and train the applications with better accessibility features. Harada et al. (2009) point out the positive attitude these users have towards taking the time to learn new tools, “a number of people with disabilities…are willing to invest more time in learning a system, especially if there are no other alternatives due to situational, monetary, or availability constraints.” Data from experiments performed by Dai et al. (2004) confirms Harada et al.’s (2009) claim. Conversely, Wobbrock et al. (2011) argue that the “burden of change” should fall on the system rather than the user, and designers should focus on ability-based design resulting in adaptive systems. Similar to Wobbrock et al. (2011), Gibson (2007) complains that “assistive technology is not always able to interpret the user interaction model” implying that the system is not adapting to the user appropriately. Until individuals with impairments become the application developers, “[a]ccommodating diverse human perceptual, cognitive, and motor abilities [will be] a challenge to every designer” (Shneiderman & Plaisant, 2010).

The driving motivation for the underlying theme of the research presented in this dissertation is recognition that more children (and adults) should have the opportunity to learn about programming and CS using these new IPEs. Improving the diversity of a user base has several advantages, as noted by Kelleher and Pausch (2005), who wrote that “[i]f the population of people creating software is more closely matched to the population using software, the software designed and released will probably better match [users’] needs.” Ladner (2008) agrees
noting that when people with disabilities are on the actual design and development teams rather than serving primarily as evaluators, the resulting application is better. Ladner (2008) describes an example where this theory holds true: a high school senior who is blind and has difficulty hearing completes her math homework using Nemeth code; however, her math teacher does not understand Nemeth code. The young girl used her programming skills to write a program that translated her math written in Nemeth code to LaTeX, which typesets math in an easily readable format. Her teacher was then able to read her math homework. Using her programming skills, she was able to solve her own problems, which is just one example of why this research is trying to empower motorically challenged children.

Many individuals suffer from a condition impairing motor functionality (NIH, 2015). If computer accessibility can be increased, new career opportunities may emerge that can help to reduce the unemployment rate among those with impaired motor function. Table 1.1 presents information collected from the National Institute of Neurological Disorders and Stroke (NIH, 2015), which lists four diagnoses impacting the motor function of children and/or young adults in the US. These individuals have the cognitive ability to understand the concepts of programming and CS, but they are unable to utilize the standard WIMP tools to interact with the computer.

<table>
<thead>
<tr>
<th>Diagnosis</th>
<th>Number Impacted</th>
<th>Age of Onset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spinal Cord Injury</td>
<td>250,000</td>
<td>56% occur between 16-30</td>
</tr>
<tr>
<td>Muscular Dystrophy</td>
<td>8,000 (males 5-24 years of age)</td>
<td>400-600 males are born with MD each year</td>
</tr>
<tr>
<td>Multiple Sclerosis</td>
<td>250,000-350,000</td>
<td>20-40</td>
</tr>
<tr>
<td>Cerebral Palsy</td>
<td>800,000</td>
<td>10,000 born with CP each year</td>
</tr>
</tbody>
</table>
A possible solution to narrow the gap of accessibility to IPEs is to use a form of a voice-driven interface to assist as an input modality for those with motor impairments. The concept of “Programming By Voice” (PBV) is the long-term focus that drives the topic of the research presented in this dissertation. Voice-driven applications, such as those presented in (Dai et al., 2004; Harada et al., 2009; Wobbrock et al., 2011), primarily cater to specific applications, like the Internet (Gibson, 2007; Trewin, 2006). These applications are discussed and compared in Chapter 3.

1.4 Dissertation Roadmap

The overarching research goal of this dissertation is to create a Vocal User Interface (VUI), named Myna, that will allow motorically challenged children to use graphical IPEs. Figure 1.8 highlights the relationship among the different components of this dissertation. Chapter 2 discusses accessibility in the classroom, and Chapter 3 focuses on existing vocal tools and related research. Chapter 4 discusses the original Myna prototype and how a user study combined with research resulted in an improved version of Myna. According to Wobbrock et al. (2011), it is important to design applications to “fit the abilities” of users rather than compensate.
for a user’s disabilities; therefore, the vocal commands should be as easy to use as possible. To accomplish this, additional user studies were performed, and these are described in Chapter 5. Chapter 6 outlines the changes that were made to Myna based on the user studies and ability-based design evaluation. Creating a VUI for one IPE (e.g., Scratch) will not resolve the issue motivated in Section 1.3. Additionally, creating a VUI for a legacy application is a time consuming process, which is highlighted in the beginning of Chapter 7. A tool to semi-automate the VUI creation process is included along with new IPEs for which VUIs were created (e.g., Scratch 2.0, Lego Mindstorms) in the latter part of Chapter 7. Chapter 8 discusses future work for this research including additional IPEs for which to apply the Myna framework (e.g., Pixly, Spherly, SNAP, Code.org), other modalities that might be considered for this type of research, and VUI/GUI testing to be performed. Chapter 9 concludes this dissertation.
In 1994, Sheryl Burgstahler of the University of Washington wrote about the need to increase the representation of people with disabilities in Science, Engineering, and Mathematics (Burgstahler, 1994). She highlighted that few persons with disabilities pursue careers in science and engineering. Flash forward to 2015, and this statement still holds true based on the most recent Bureau of Labor Statistics data where only 2.4 percent of the US disabled population are employed in Computer and Mathematical occupations (Bureau of Labor Statistics, 2013). While the unemployment rate for those with disabilities (13.2 percent) was almost double that of those without disabilities (7.1 percent) in 2013, the percent of the US disabled population employed in Computer and Mathematical occupations is almost equal (2.4 percent disabled versus 2.8 percent not disabled) (Bureau of Labor Statistics, 2015). Therefore, as Burgstahler (1994) points out, “students with disabilities can find success in science, engineering, and mathematics fields.”

However, there are three primary obstacles discouraging persons with disabilities from studying science, engineering, and math: preparation, access, and acceptance (Burgstahler, 1994). Students with disabilities are often not prepared to study science, engineering, and math (now referred to as STEM: Science, Technology, Engineering, and Math) in higher education due to not taking the proper courses in high school; students with disabilities do not generally have access to the necessary technology, labs, and resources necessary to study these fields; and professors have a reluctance to work with students with disabilities (Burgstahler, 1994). The key
solution to these three obstacles, which Burgstahler (1994) reiterates several times, is for students with disabilities to start using technology as early as possible, which is what the work presented in this dissertation expects to encourage.

This chapter discusses technology improvements over the past two decades and what that means for the classroom. How technology has improved for persons with disabilities from 1992 to 2008 is discussed in Section 2.1. A brief review of how technology is being used in classrooms is given (Section 2.2) followed by a review of how technology is being used in the Special Needs classroom (Section 2.3) as well as first-hand accounts of technology in the Special Needs classroom (Section 2.4). Section 2.5 concludes this chapter.

2.1 Computers and People with Disabilities: Then and Now

In a 1992 issue of the Communications of the ACM, Glinert and York (1992) made a plea for more “research and development on technologies for people with disabilities” entitled “Computers and People with Disabilities.” This article described a few research projects and entities serving those with disabilities; however, Glinert and York declare that this work is not nearly enough. They point out that the number of persons with disabilities is higher than one might think, highlighting that (in 1992) “over half a million Americans are legally blind” and 100,000 out of nearly five million US scientists and engineers have a physical disability. This point is made even bolder by quoting an oral communication made by Brown, “we are all disabled, it is just a matter of degree.” With the launch of the ACM Transactions on Accessible Computing (TACCESS) in 2008, TACCESS reprinted the “Computers and People with Disabilities” article along with reactions about how assistive technology research has changed from the four primary authors cited in the original work: Edwards, Newell, Vanderheiden, and Ladner. Their reactions are summarized here.
The overwhelmingly obvious difference between then (1992) and now (2008) is the Web. The Web is not mentioned in the 1992 article, but the Web was then only a glimmer of what it is now. Edwards mentions how in 1992 accessibility research was trying to play “catch up” with the “mainstream,” but in 1992, researchers were trying to keep up with the GUI, and now researchers are trying to keep up with the Web (Edwards, 2008). Vanderheiden (2008) echoes Edwards stating that “significant challenges remain” on making the Web accessible. The primary barrier that Vanderheiden (2008) points out is for most users, the Internet can be freely accessed from public computers; however, users needing special assistive devices cannot simply visit a public library to use the computer. Moreover, it is far too expensive and complicated to expect all public computers to possess the full range of assistive devices (Vanderheiden, 2008). Ladner (2008) also indicates that the Web is a “major challenge,” particularly with the advent of Web 2.0 increasing the number of Web contributors to the “hundreds of millions to billions.”

Glinert and York (1992) remark that computer and information access “by disabled people has been hampered by short sightedness on the part of computer and communication system designers.” Newell (2008) has found this statement to remain true. While much has changed and improved, with new technology comes new challenges (Vanderheiden, 2008). Edwards (2008) points out that the legislation mentioned in the 1992 article is still in force and has been enforced. Legislation continues to be updated to ensure the needs of those with disabilities are met, indicating an awareness of persons with disabilities. However, the fact that legislation is necessary demonstrates the lack of accessibility in technology. Edwards (2008) explains that “[i]f systems were designed from the start to be accessible, there would be no need to develop new technologies to make them so or to have laws mandating the application of these technologies.” Designing systems from the beginning with accessibility in mind is a focus of
universal design, which is increasing in practice not only because of legislation, but because the market demands it (Ladner, 2008; Vanderheiden, 2008). While universal design is beneficial, it does have its disadvantages, especially when treated as an add-on (Newell, 2008). Newell (2008) prefers the design methodology of looking at a target audience as people rather than a set of characteristics and for the designers to “develop empathy with their user group” (Newell, 2008). Moreover, designers should think more about what users might want versus thinking only about what users might need (Newell, 2008).

Ladner takes user-centered design to the next level imploring developers to design for “user empowerment” with the ultimate goal of “crea[t]ing tools that enable persons with disabilities to solve their own accessibility problems” (Ladner, 2008). It is for this reason that it is imperative that more students learn to program at an early age, particularly, those with disabilities. However, in order for students with disabilities to learn how to program, they must be exposed to the tools and be able to access them.

2.2 Technology in the Classroom

As mentioned in Chapter 1, many instructors are utilizing block-based programming environments to teach computer programming due to the environments’ level of creativity sparking student interest. The following research reviews examples of how these environments are being taught (Section 2.2.1), how one specific platform, App Inventor, is being used in the classroom (Section 2.2.2), and how these environments help to teach problem-solving (Section 2.2.3).

2.2.1 Teaching Graphically with Blocks

Block-based programming environments are being taught worldwide, and there are many initiatives pushing the teaching of these environments, such as the College Board’s new CS
Principles Course (College Board, 2015). This course encourages block-based programming environments in addition to other techniques (e.g., CS Unplugged) to motivate student learning (Gray et al., 2014). While CS Unplugged is not graphical, it does involve the instruction of CS concepts in unconventional ways, namely, without a computer (Bell et al., 2009). CS Principles instructors can choose the IPE they wish to use; however, App Inventor and Scratch are popular choices based on provided curriculum (CS Principles, 2015).

Outside of the K12 classroom, many universities host summer camps for elementary, middle, and high school students. At the University of Alabama, summer camps in the past have consisted of three weeks of CS curriculum for high school students (Table 2.1). Each Spring, students from the University of Alabama also visit elementary, middle, and high schools in the city of Tuscaloosa, Alabama to teach Lego Mindstorms robotics. As another example, the Colorado School of Mines has also offered summer camps focusing on a free camp for economically challenged families (Miller et al., 2005). This one-week camp was comprised of web design, Lego Mindstorms robotics, GPS scavenger hunts, creating video games, and animation (Miller et al., 2005). Yet another summer camp, Comp Camp, offered by Auburn University, introduces students to Alice and Lego Mindstorms, and a second camp, Robo Camp, is offered to advanced students who performed well during Comp Camp (Marghitu et al., 2009).

<table>
<thead>
<tr>
<th>Camp</th>
<th>Pre-requisites</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1: Intro to Java</td>
<td>No experience needed; Grades 9-12</td>
<td>Taught intro to Java using Greenfoot</td>
</tr>
<tr>
<td>Week 2: Robotics</td>
<td>Existing knowledge of Java or attendance of the first camp</td>
<td>Taught Java using Lego NXT and Lejos</td>
</tr>
<tr>
<td>Week 3: Android App Inventor</td>
<td>Existing knowledge of Java or attendance of the first camp</td>
<td>App Inventor Block Language and App Inventor Java Bridge</td>
</tr>
</tbody>
</table>

Table 2.1. University of Alabama summer camp schedule.
The underlying theme across these four examples of university outreach activities is the use of block-based programming environments.

2.2.2 Teaching with App Inventor, a Block-Based Example

The popularity of mobile devices has inspired much interest as a context for teaching computation (Mahmoud, 2011). In fact, a new learning model has emerged, LOCAL (Location and Context Aware Learning) (Barbosa et al., 2008), which combines mobile devices and wireless networks to create a new learning context. In this section, we summarize a portion of the work that has emerged in using smartphones as a context for teaching CS.

At the University of San Francisco (USF), Wolber has integrated App Inventor into a general education course over the past three years and reported that App Inventor was “his most satisfying teaching experience in seventeen years” (Wolber, 2011). App Inventor is a visual programming language that allows users to write apps using a block-oriented drag-and-drop interface to create both the user interface of an app, as well as to specify the app’s behavior and functionality. An emulator is available for App Inventor so that apps can be executed on a local desktop. App Inventor also integrates with Android smartphones and tablets, which enables the user-made applications to be tested on a physical device. Before teaching App Inventor, Wolber’s course used Lego Robots (Lego Mindstorms, 2015) and Media Computation (Guzdial & Ericson, 2009). The course focuses both on programming and the real-world impact of mobile applications (Wolber, 2011). Because students were able to start building applications immediately with App Inventor, they were motivated as the semester progressed to learn “how to solve hard logic problems.” Although the students attending the course were not CS majors, they successfully learned how to solve problems and, more importantly, felt empowered at the conclusion of the course (they even presented along with Senior and Master’s project students at
USF’s annual CS Night. There were 11 students out of 41 total who went on to take the next course in the CS sequence. In the past, this CS0->CS1 bridge has been very rare, because the students that enroll in CS0 often take it because they have little confidence in mathematics.

As described in (Fenwick et al., 2011), Fenwick and Kurtz (from Appalachian State University) and Hollingsworth (from Elon University), taught their senior classes using the Android SDK (Eclipse, 2015) and App Inventor. The Elon University course started as a lecture-based course with hands-on activities, which was followed by a project-based approach. The Appalachian State University course was project-based from the start. Both universities found that “students enjoyed the course” and students were once again exhibiting an “entrepreneurial and independent spirit” (Fenwick et al., 2011).

To assist in teaching the importance of human-computer interaction, Loveland used App Inventor to motivate students (Loveland, 2011). At the conclusion of the course, one student commented: “It is cool that the course applied the latest technologies in software development to mobile and web design.” This statement summarizes why (Barbosa et al., 2008; Fenwick et al., 2011; Goadrich & Rogers, 2011; Mahmoud, 2011; Wolber, 2011) are taking advantage of mobile computing as a teaching context. Students are able to learn about programming using tools that are very personal and applicable to their daily lives.

A few college courses on mobile computing have been described above; however, these courses focused on college-level students in beginning or non-major programming courses. Roy (2012), from Valdosta State University, organized summer camps for high school students using App Inventor in 2011 and found it to be an effective tool for novice programmers due to the visual environment being similar to Scratch. Moreover, Roy (2012) noted that transitioning from
App Inventor to Java would be easy due to the Java Bridge functionality, but he does not provide an assessed study.

2.2.3 Problem-Solving Using Blocks

There exists a large volume of work regarding the advantages and disadvantages of using a visual programming language to increase problem-solving skills. In 1983, Pea (1983) conducted a year-long study using Logo to teach critical thinking to 8-9 and 11-12 year old students. Pea found that using Logo did not increase the students’ ability to solve problems (Pea, 1983). Instead, he found that students were exhibiting “production without comprehension” and that students continued to struggle with logic (Pea, 1983). More recent research, however, contradicts Pea’s findings. Carlisle et al. (2005) taught three semesters of an introductory programming course: one semester was taught using Ada and MATLAB, and the other two semesters were taught using RAPTOR, which is a Rapid Algorithmic Prototyping Tool for Ordered Reasoning created by Carlisle et al. (2005). In previous offerings of this course, students were given the opportunity to use Ada, MATLAB, or flowcharts to express algorithms, and students typically preferred a visual representation to a textual representation motivating the development of RAPTOR (Carlisle et al., 2005). Carlisle et al. (2005) compared the final exam across the three semesters and found that, overall, students performed better using RAPTOR than with textual languages like Ada and MATLAB.

Whitley and Blackwell (2001), and Green and Petre (1996), argue that visual programming languages are beneficial and can allow for increased efficiency over textual languages. In support of visual programming languages, Whitley argues that problem-solving is easier with increased organization, and visual programming languages allow for more organization; however, visual programming languages can be poorly designed or mismatch the
problem to be solved (Whitley, 1997). Green and Petre (1996) suggest that the Human Computer Interaction (HCI) aspects of visual programming languages are in need of further development. After presenting both advantages and disadvantages along with the lack of empirical evidence supporting visual programming languages, Whitley concludes with a call to action to either continue developing visual programming languages with the hope they evolve in “appropriate directions” or to utilize empirical testing to ensure the visual programming language is well designed (Whitley, 1997).

Resnick appears to have followed the latter strategy suggested by Whitley (Resnick, 2007; Resnick et al., 2009) in that he studied how children in kindergarten learn, and as a result, the MIT Media Lab developed the Scratch IPE. Scratch allows students to imagine, create, play, share, and reflect (Resnick, 2007) by following three critical design principles: be tinkerable, be meaningful, and provide a social environment (Resnick et al., 2008).

2.3 Technology in the Special Needs Classroom

As Section 2.2.1 mentioned, block-based, visual programming environments are being taught during summer camps for typically developed students; however, these fun, creative environments are not being taught in special needs classrooms. Instead, students are learning more basic computer literacy skills (Marghitu & Zylla-Jones, 2006; Marghitu et al., 2009), which is important and valuable. These literacy skills are being taught in a fun way using Train and Assess IT (TAIT), a web-based tool (Marghitu & Zylla-Jones, 2006). Results from the Auburn camp demonstrate that teaching with TAIT was successful, and students with disabilities increased their computer knowledge in addition to improved memory and communication. Other programs that students with special needs use while at Comp Camp at Auburn University include
Beyond computer literacy, there are two programs that provide students with disabilities the support necessary to pursue careers in STEM fields. First, there is the High School High Tech program, which is available in states around the US (NCWD, 2015; Nietupski et al., 2001). It is a program for high school students with mild disabilities (most have learning disabilities while a small percentage has “hearing impairments, physical disabilities, Aspergers Syndrome, behavior disorders, Attention Deficit Disorder or Attention Deficit Hyperactivity Disorder, and or cognitive disabilities”) where the students learn about technical fields through a variety of activities consisting of tours to technical companies, shadowing employees at technical companies, participating in internships, and summer technical camps (Nietupski et al., 2001). Iowa found success in this program as 21 out of 36 participants went on to post-secondary education; 14 of those elected to study a technical field (Nietupski et al., 2001). Unfortunately, Iowa no longer offers this program, but it is available in ten states (NCWD, 2015). Second, there is the DO-IT (Disabilities, Opportunities, Internetworking, and Technology) program at the University of Washington, which was started in 1992 (Burgstahler, 1994). This program (still in existence (DO-IT, 2015)) aims to break down the barriers for students with disabilities described previously: preparation, access, and acceptance through internetworking, mentoring, summer programs, and special projects (Burgstahler, 1994).

2.4 Getting Involved

In 2012, the Computer Science Collaboration Project (CSCP) offered Mini-Grants to researchers wanting to share CS to a more diverse population in which the University of Alabama, Auburn University, and United Cerebral Palsy partnered (CSCP, 2012). Students
with disabilities worked side-by-side with typically developed students in fun, interactive computer science and robotics activities at three separate events. It was very important to be as hands-on as possible in order to maintain the participants’ attention. It was also important to ask many questions to keep the participants engaged and focused.

At two of the events, both Lego Mindstorms and App Inventor were introduced. The first segment involved programming a Lego Mindstorms robot. Two different demonstrations were performed, and the participants were asked to help write the subsequent programs to perform tasks such as dance to music and follow a line. By asking the participants to write the program, they were able to see how trial and error can be used to solve problems, and they were able to see success in their efforts. App Inventor was then demonstrated, and the students were asked to help complete simple programs (e.g., “HelloPurr,” “PaintPot” from (Wolber et al., 2011)). Figure 2.1 is an image of a participant completing an App Inventor program. These two events were then concluded with ideas of what students could do by studying CS. A third event was hosted by Auburn University where Kodu, Alice, and Lego Mindstorms robots were taught. Students worked together in teams and then presented their work to their parents and siblings at the conclusion of the event.

The overall goal was to increase CS awareness among youth with disabilities wanting to demonstrate to the participants that there are numerous opportunities and loads of fun to be had in the field of CS. Although not the target audience, parents and siblings heard the same information and were made aware of what Computer Science is and what can be accomplished with CS knowledge. The participants enjoyed the workshops based on their reactions throughout the sessions and feedback provided. One student sent an email stating that he now wants to pursue more classes in CS, particularly summer camps offered by local universities.
During the summer of 2013, the United Cerebral Palsy of Greater Birmingham (UCP) hosted a summer camp for children and young adults, ages 14 to 23, with special needs. This is an annual summer camp consisting of social opportunities for attendees, arts and crafts activities, physical activities (i.e., basketball, volleyball, bubbles, water balloon fights, and yoga), field trips, and games. There were six counselors and approximately 50 high school volunteers who worked various hours during the six-week camp. Throughout the summer, various computer-based activities were offered for interested attendees.

The camp attendees were diverse, and the range of special needs varied between attendees. Some attendees had minor cases of Asperger's or learning disabilities with little to no physical limitations. Other attendees had severe Autism, Down Syndrome, or physical disabilities limiting him/her to a wheelchair and unable to feed him/herself.

There were three primary motivations for attending the camp.

1. The primary motivation for volunteering at this summer camp was to evaluate Myna, which is described in detail in Chapter 4, and the evaluation is discussed in Chapter 5.

2. The secondary motivation for volunteering at this summer camp was to learn more about the intended audience of Myna. Did the attendees have an innate curiosity about computers as demonstrated in CS camps taught elsewhere?

3. The final, and most important, motivation was to share the joy of CS through fun activities such as CS Unplugged activities and programming with a new audience.
By attending every day during the first week of camp and participating in activities like basketball, yoga, and coloring, a rapport was developed with the camp participants. Throughout the six weeks of camp, three different Computer Science activities were introduced to the attendees, and because of the rapport, the campers were more comfortable either participating or declaring their disinterest in the various activities.

1. The first activity was conducted to spark interest in CS through a fun game of “programming” a person to make a Peanut Butter and Jelly sandwich. Nine of 22 attendees participated, and the activity provided occasions for amusement thereby engaging the participants who did a great job of giving explicit instructions (Figure 2.2).

2. The second activity was to create programs using Scratch and Myna. Since there was only one computer, this was a one-on-one activity and fewer attendees chose to participate (three out of 22).
3. The third activity was programming the Lego Mindstorms robot. The participants did this in small groups despite only having one computer. There were five out of 22 attendees who elected to participate.

During the six weeks of camp, several observations were made. Firstly, there was an overwhelming lack of participation. During the camp, the attendees preferred to spend time being social. The attendees of this camp form tight bonds with one another and do not often get to spend time interacting with each other. Secondly, when attendees were asked about their level of computer usage, the two responses given were:

1. Using the Internet (watching YouTube videos); and
2. Homework.

Those who participated in each activity genuinely enjoyed it, but the majority of the attendees did not have enough experience with computers to have an interest in learning more. Lastly, more physical activities were preferred over sedentary activities. The attendees loved being in the gym or outside. The Lego robot was slightly more popular than Scratch because this was a more social activity, and it required the participants to move around a bit more. It was also entertaining for others to observe.

Working with this population in the future, computer-based activities should be more social in nature. Moreover, as a general thought, these students should have more access to computer programs beyond the Internet, and the classroom is a great place to start.

2.5 Conclusion

Burgstahler (1994) identifies three barriers preventing students with disabilities from studying science, engineering, and math: preparation, access, and acceptance. Her primary solution to overcome these barriers is for students to begin using technology at a young age, but
students with disabilities cannot always freely access technology equivalent to those without disabilities (Vanderheiden, 2008). Moreover, the software available on the computer can be a hindrance in and of itself as persons with disabilities may need assistive devices to interact with the software; the main problem being that designers do not always design with users’ disabilities (or abilities – what users can do) in mind (Glinert & York, 1992; Newell, 2008). In order to truly rectify this dilemma, persons with disabilities need to become the developers (Ladner, 2008), which requires that persons with disabilities be taught how to program. Based on hands-on activities conducted with students with disabilities (cognitive and physical), students with disabilities first need more experience and exposure to computers, which programs like Comp Camp are doing (Marghitu et al., 2006; Marghitu et al., 2009).

Beyond computer literacy, students with disabilities need to have access to IPEs (e.g., Scratch, Lego Mindstorms), but these applications are heavily dependent on the WIMP metaphor. The next chapter discusses background research on how speech recognition could be of use in providing accessibility to IPEs for those with motor impairments.
CHAPTER 3

BACKGROUND RESEARCH

GUIs require heavy usage of the keyboard and mouse for input. As described in Chapter 1, this limits motorically challenged users. A suggested solution is to provide vocal input, which has proven to be successful (Martin, 1989; Jung et al., 2007; Hauptmann & Rudnicky, 1990). Tools such as the Vocal Joystick (Harada et al., 2009) allow users to work within GUIs vocally. While tools such as these are advantageous and users find them beneficial (Harada et al., 2009), they may not be as beneficial when manipulating objects within a GUI such as in an IPE (e.g., Scratch). IPEs require navigation in addition to the manipulation of objects on the screen.

The key to creating a successful, universally usable tool is to apply an ability-based design, in which developers strive to take advantage of what abilities the user possesses and to make the system adapt to the user, rather than make the user adapt to the system (Wobbrock et al., 2011). This should be a general design strategy for systems/tools/applications for all users, not just those with impairments. To ensure a design is ability-based rather than disability-based, Wobbrock et al. (2011) developed seven principles: Ability, Accountability, Adaptation, Transparency, Performance, Context, and Commodity (see Section 3.1 for a detailed explanation of these seven principles).

Martin (1989) researched two claims regarding the validity of speech as an input device: 1) Speech is faster than typing; and 2) Speech increases productivity. Martin’s literature search proved claim number two and some of the research validated claim number one, but with some
uncertainty. Martin created an experiment to test each claim by having users navigate a graphical application. The resulting data proved both claims; speech had a 108% time advantage versus typing full-word commands, and speech increased productivity by allowing the user to reduce the amount of glances at the keyboard and providing an “additional user-response modality” (Martin, 1989). Jung et al. (2007) and Hauptmann and Rudnicky (1990) performed comparisons of vocal versus keyboard input. Jung et al. (2007) presented a brainstorming experiment and found that utilizing voice in a group setting to collect ideas resulted in a larger quantity of higher quality ideas as compared to typing ideas. Hauptmann and Rudnicky (1990) had users enter a series of numbers three different ways: 1) Using voice only; 2) Using voice to enter the numbers and a keyboard for error correction; and 3) Using a keyboard only. Users performed fastest using the second methodology (multimodal) with the first methodology (voice only) less than a tenth of a second slower, and the third methodology (keyboard only) being the slowest at about one second slower. Based on the results presented by Martin (1989), Jung, et al (2007), and Hauptmann and Rudnicky (1990), voice is not only a viable input modality, but it has the potential to allow the user to be more efficient.

3.1 Ability-Based Design

Wobbrock et al.’s (2011) definition of *ability-based design* involves seven principles (see Figure 3.1): ability, accountability, adaptation, transparency, performance, context, and commodity. These seven principles are broken up into three different categories. Firstly, ability and accountability, which are both required principles, are categorized as “stance” meaning that designers should take the stance with the perspective of what users “can do” rather than what they “cannot do” or what all users can do (Wobbrock et al., 2011). Secondly, the adaptation and transparency principles, which are recommended, belong to the interface category because of
Figure 3.1. Seven principles of ability-based design (Wobbrock et al., 2011).

their relationship to an application’s interface. Lastly, performance, context, and commodity belong to the system category as these three principles relate to the system as a whole. The seven principles are discussed in detail below.

1. Ability: As previously mentioned, this is the idea that designers focus on users’ abilities rather than disabilities (Wobbrock et al., 2011). Ladner (2008) echoes this idea citing that interface design should focus on what users can and want to do. The goal should be to make “technology fit people” (Kelley, 2007). Regarding designing for everyone, this is an impossible task; instead, designers should try focusing on smaller groups of users, characteristics about whom designers understand and for whom designers have empathy (Newell, 2008).
2. Accountability: The application should be held accountable for carrying the “burden of change” (Wobbrock et al., 2011). User should not be held responsible to adapt any part of themselves (e.g., bodies, knowledge, behavior) to an application (Ladner, 2008; Wobbrock et al., 2011).

3. Adaptation: This is the primary principle that removes the need for users to adapt themselves to an application, and instead, requires that the application adapts to the user without special add-ons (Wobbrock et al., 2011). Often, this adaptation is presented via “user-selectable options” empowering the user to customize the application to his/her preferences (Ladner, 2008). Ladner (2008) considers the iPhone’s VoiceOver application (Apple’s version of a screen reader for the iPhone allowing blind users to visualize the phone’s screen) a great example of an adaptable program. He explains that while most designers would use a natural, conversational rate of speech for the application, most blind users would prefer an increased rate of speech, and VoiceOver allows the user to alter the rate of speech to his/her preference (Ladner, 2008).

4. Transparency: Applications should provide a level of transparency to the user meaning that not only should users be able to “inspect, override, discard, revert, store, retrieve, preview, and test” various adaptations, but adaptable settings should always be visible to the user despite any automatic updates made to the application (Wobbrock et al., 2011). Moreover, features increasing adaptability or accessibility should be built into the infrastructure of all devices providing access to ubiquitous devices for all users (Vanderheiden, 2008). As an example of a transparent device, Wobbrock et al. (2011) describe Trewin and Pain’s (1997) dynamic keyboard model, which makes suggestions to the user (e.g., “Sticky Keys”) rather than making changes behind the scenes.
5. Performance: Wobbrock et al. (2011) recommend this principle but note that hardware and software may not be capable of doing so just yet; however, if possible, designers should attempt to “sense, monitor, measure, and model” users’ actions during runtime. Microsoft Word’s function to correct the spelling of words as a user types is an example of the performance principle because Word examines a user’s performance during use.

6. Context: The same as with performance, Wobbrock et al. (2011) recommend this principle but note that technology may not be ready to fully meet this design principle. As much effort as possible should be focused on “sensing” a user’s performance and “anticipating” changes based on the context of the user’s interactions (Wobbrock et al., 2011). This appears to be a more difficult principle to accomplish; of the 14 applications that Wobbrock et al. (2011) review, only two exemplify context. An interesting example is walking user interfaces, which detect whether a user is walking or standing still, and if the user is walking, the interface changes increasing the size of the content displayed on the screen (Wobbrock et al., 2011). Another example would again be Microsoft Word, which not only corrects the spelling of words, but also points out possible grammatical errors based on the context of the sentence.

7. Commodity: Wobbrock et al. (2011) presents a strong case for utilizing low-cost technology, “cost, complexity, configuration, and maintenance of specialized hardware and software are perpetual barriers” for technology use. Additionally, they observed that less than 60% of users needing “access technologies” actually use them. Some of these devices can cost thousands of dollars, and it may be unreasonable for the users to be required to spend that much money over and above any medical costs they have. Voice-driven tools offer a modality for input that is often cost effective and addresses the needs of a broad range of users who have motor challenges, but are still cognitively capable and have an ability to
speak. Vanderheiden (2008) mentions the idea of “alternate interfaces” that can plug into ubiquitous devices, the software on which is constantly updated. By keeping the software updated, the lifespan of the device, and the adaptable interface, will lengthen, thus, decreasing costs (Vanderheiden, 2008).

While it may not be possible to include all seven of these design principles within an application, it is important to attempt to do so. In Chapter 5, Myna is evaluated based on these design principles, which while not considered in the design of early Myna (Chapter 5.1), they were considered for later versions (Chapter 5.3).

3.2 Programming by Voice

Programming by voice (PBV) is not a radical concept. Prior efforts, exemplified by (Begel, 2005; Dai et al., 2004; Désilets et al., 2006; Harada et al., 2009; Hubbell et al., 2006; Raman, 1996; Shaik et al., 2003), present some form of a voice-driven application. Begel (2005), Désilets et al. (2006), Hubbell et al., 2006, and Shaik et al. (2003) discussed voice-controlled applications for textual programming (Section 3.2.1), whereas Dai et al. (2004) and Harada et al. (2009) focused on voice-driven cursor control techniques. These voice-driven cursor control techniques along with speech-based interface interaction (Raman, 1996) are described in Section 3.2.2.

3.2.1 Voice Programming for Textual Languages

The goal for Begel (2005) was a design based on “natural verbalization.” The problem that both Begel (2005) and Désilets et al. (2006) experienced is that code and natural language are far different, and algorithms had to be created to provide the required flexibility a user would need, particularly since unlike natural languages, programming languages are not spoken (Begel
& Graham, 2005). Begel (2005) provided an illustrative example that highlights this challenge within the context of verbalizing a common loop structure:

```java
for (int i = 0; i < 10; i++) {
}
```

If a programmer were to speak this statement aloud, it might be spoken as:

“for int i equals zero i less than ten i plus plus”

As a first effort, this appears to be a reasonable way to state the loop structure, and it is quite probable that every programmer says this in his/her head while typing a “for” loop. Unfortunately, this statement has two problems: 1. The brackets and punctuation are not dictated, and 2. Some of these terms are unusual for a typical natural language processor. When speaking code, there are three factors to consider:

1. Most words used in programming have homophones (e.g., four and for, c and see, I);
2. Capitalization is not articulated, but programming languages are case sensitive; and
3. Spaces are not clear (i.e., a variable name such as numberParameters would be interpreted as “number parameters”) (Begel & Graham, 2005).

Instead of the above verbalization, Begel (2005) explains that, due to the presence of homophones, the voice recognition engine may understand the words spoken as:

“4 int eye equals 0 aye less then ten i plus plus”

Although Begel solves some of the issues, particularly the punctuation, there is still some responsibility placed on the programmer to learn the structure and terminology for Spoken Java, which allows a programmer to write programs vocally (Begel & Graham, 2005). Spoken Java is the language used when coding verbally in SPEED (for SPEech EDitor), an Eclipse plug-in (Begel & Graham, 2006). In a small user evaluation of SPEED consisting of five participants, Begel and Graham (2006) found that in the first session, the speed of the system and number of
speech recognition errors that occurred as a result was frustrating to the participants and they were unable to finish the code required for the task. However, the second group used a human as the speech recognition engine, and the results were far better, highlighting that if the speech recognition engine were more accurate, users would be more successful. Most users said that they would not use vocal programming in their daily work unless they worked from home where the room would be quiet, and they would not interfere with the work of others (Begel & Graham, 2006).

Désilets et al. (2006) discovered the same natural verbalization issue and solved it in a similar manner. Désilets et al. (2006) created VoiceCode, a system that allows users to program by voice through the use of a simple grammar, which is then translated to proper syntax. There is a proper grammar the user must learn; however, this same grammar translates to any programming language. Désilets et al. (2006) also interjected code navigation and error correction abilities in addition to adding context-sensitivity. For example, if the user makes an utterance for “number parameters,” VoiceCode looks first for corresponding possibilities (e.g., previously declared variables, methods, or classes). If there are no matches, VoiceCode creates a new symbol and uses the context surrounding the location to determine capitalization (i.e., NumberParameters if a class name is expected or numberParameters if a variable is expected).

Hubbell et al. (2006) implemented a different approach than (Begel, 2005; Begel & Graham, 2005; Begel & Graham, 2006; Désilets et al., 2006) and used an interface approach in their system called VASDE (Voice-Activated Syntax-Directed Editor). VASDE is only written for Java and uses Eclipse as the underlying platform. Command-based navigation (i.e., the user states a one word action such as “select” followed by a label) is used to control the interface. While the interface is vocally controlled, only a portion of the syntax is entered vocally. Java
expressions have not yet been implemented; therefore, these must be entered via the mouse/keyboard. VASDE implements dialog windows to complete various constructs such as a for loop where the user only needs to type in the expressions (e.g., the initialization, the evaluation, and the increment) of the for loop.

Yet another solution to PBV is the SpeechClipse tool, in which Shaik et al. (2003) utilized a rule-based grammar in coordination with Java’s Robot class to translate the commands recognized by the speech recognition engine into activities performed by the keyboard. In their approach, as the user speaks, the mouse or keyboard performs actions based on the commands (i.e., the mouse and keyboard are programmatically controlled by the Java Robot class, which is driven by parsed words spoken by the user). The utterances are first compared to the grammar, and then, these tokens are mapped to either a keyboard event or an Eclipse command. Once the proper mapping is made, the appropriate action is performed (Shaik et al., 2003).

3.2.2 Speech-Based Interface Interaction

Beyond coding, there is a necessity for manipulating the user-interface of any Integrated Development Environment (IDE). Dai et al. (2004) and Harada et al. (2009) focused on cursor movement. Dai et al. (2004) evaluated a voice-driven math program for the visually impaired, but experienced the same issues as (Begel, 2005; Begel & Graham, 2005; Begel & Graham, 2006; Désilets et al., 2006; Hubbell et al., 2006; Shaik et al., 2003). Dai et al. (2004) used a grid-based solution to allow the user to identify where on the screen he/she wishes to click. The grid begins as the entire screen is divided into nine sections. The user selects in which of those nine segments he/she wishes to click by verbalizing the number. That portion of the screen is then partitioned into nine sections, and again the user verbalizes the section in which he/she wishes to click. This process repeats three times until the specific item of interest is identified. Dai et al.
(2004) conducted a small study that compared whether a grid-based system works best with nine cursors or with one cursor; the difference being that with nine cursors, the user makes three verbal clicks, and with the one cursor solution, the user makes four verbal clicks. The nine cursor solution was determined the fastest method.

The specific approach explored by Harada et al. (2009) used a very different method for cursor control. Rather than use words such as “move right, move right, stop,” they identified ten syllable sounds that are mapped to direction and speed (Figure 3.2). The syllables are selected based on the International Phonetic Alphabet (IPA). Of the ten syllable sounds, eight are vowel sounds, and they “were chosen because they represent the eight most distinct sounds in the vowel map periphery that are present in as many of the world’s major languages as possible” (Harada et al., 2009). A disadvantage of using the syllable mapping is that it is not instinctive and some syllables are difficult to pronounce at first; therefore, it requires some training. The actual tool, called the Vocal Joystick, was developed with the goal of user-friendliness. It provides multiple methods of feedback for the user, which is particularly helpful in the training sessions. If a certain phonetic sound is difficult for a user to pronounce, the system can be modified for that user. Additionally, it is built in such a way that applications can be developed around it such as VoiceDraw, which is supported by Wobbrock et al. (2011) as meeting three of their seven guidelines: adaptation, performance, and commodity. The most important aspect of the Vocal

![Figure 3.2. Vocal Joystick vowel mapping (Harada et al., 2009).](image-url)
Joystick, and thus VoiceDraw, is the ability to adapt to the user rather than the user having to adapt to the system.

Interface interaction is not confined solely to input. It is equally important to understand how output can be conveyed. There are various screen readers available (Smith, 2007; Thatcher, 1994); however Emacspeak is an order of magnitude more advanced than a typical screen reader (Raman, 1996). Emacspeak is an audible output interface for Emacs with extensions for any Emacs application. Raman (1996) developed Emacspeak with the intention of having the applications speak rather than the screen being read. Typical screen readers do not have access to the underlying application; they are top-level and merely vocalize the content on the screen. Emacspeak, however, is at the application-level and has access to the same data to which the visual display has access thereby “[treating] speech as a first-class [input/output] medium” instead of an after thought (Raman, 1996).

3.3 Vocal Limitations

Several papers in the area of PBV (Begel, 2005; Begel & Graham, 2005; Begel & Graham, 2006; Dai et al., 2004; Désilets et al., 2006; Harada et al., 2009; Hubbell et al., 2006; Raman, 1996; Shaik et al., 2003) demonstrate that voice can be a reasonable alternative to the mouse and keyboard. However, an undesirable consequence is the potential for vocal strain. Haxer et al. (2001) reported a case study of a doctoral candidate/lecturer who began experiencing pain due to tendonitis and decided to utilize a voice recognition program instead of her keyboard. She used voice recognition approximately four to six hours daily. Due to the heavy usage, she began to experience vocal strain and vocal fatigue. She then went to the Multidisciplinary Voice Clinic of the University of Michigan Vocal Health Center for evaluation. After performing several tests, it was determined that her vocal quality and sound were below average. She then
started speech therapy where she was taught proper vocal techniques such as warming-up her vocal chords before speaking for long periods of time and hydrating periodically during vocal use. After learning how to properly exercise her vocal chords, she began using voice recognition again for four to six hours daily, and she no longer experienced vocal strain or fatigue. Moreover, she was re-evaluated, and her vocal quality and sound had improved to normal levels. De Korte and Van Lingen (2006) performed a study to determine if voice recognition in the workplace could improve posture and productivity and how user friendly this technology was perceived among participants in the study. While voice recognition did improve posture, most participants found they were more productive, and most participants found the voice recognition to be user friendly; five of the 15 participants complained about sore throats as a result of the frequent speaking.

Another limitation of voice as an input is speech recognition errors. Oviatt (2000) mentions that errors primarily occur when a user alters his or her voice and articulates words differently than when the user trained the speech recognition system. Oviatt (2000) also observed that users’ articulation alters with changes in emotional states, the surrounding noise, and the tasks the user is performing at the time. She proposes using multimodal input to reduce these errors and to increase the efficiency of correcting speech recognition errors. A multimodal approach for Myna is left as an area of Future Work; however, by understanding the primary reason for errors occurring, users can work to speak consistently, thereby reducing the error rate.

3.4 Additional Input Modalities

Gestures and eye tracking are other possible input modalities to consider. Intel envisions controlling the computer by hand gestures and facial expressions (Gaudin, 2013). Bolt (1980) also proposes using gestures to control the computer. Busjahn et al. (2014) discuss the
possibilities of using eye tracking in programming, particularly in programming education. After conducting a case study of using eye tracking to program, Busjahn et al. (2014) suggest that in the realm of CS education, eye tracking could be used to identify students’ challenges, code comprehension, and understanding how a user writes and debugs code among other topics.

Multimodal interaction is another popular solution (Alsuraihi & Rigas, 2007; Bolt, 1980; Cohen et al., 1998; Neto et al., 2008), which allows the user to interact with his or her voice and a secondary device such as a pen, keyboard, mouse, or gestures. Alsuraihi and Rigas (2007) present an empirical study involving multimodal interaction where the user interacts by speaking but may use visual or aural modalities for “reading” the interface. Two interface design toolkits were created for user testing: a typical GUI and a multimodal GUI (a GUI with text-to-speech vocal input). The results showed that “[t]he use of voice-instruction as a way of interaction was found to be more effective than the user of several visual interaction metaphors” (Alsuraihi & Rigas, 2007). Similarly, Bolt (1980) describes a media room where a user manipulates objects on the screen using both voice and gestures. This proved to be a convenient method of interaction because the gestures allowed for more enhanced understanding of the user and more natural commands delivered by the user. Neto et al. (2008) presented a multimodal interface for the web, which would “allow users to select and switch between input modes avoiding physical overload in the use of one modality” supporting that there are limitations using any one modality. Not only can multimodal interaction provide a more convenient method of interaction, but it can also provide a more efficient means of interaction. Hauptmann and Rudnicky’s (1990) work demonstrates that a multimodal approach is faster than speech or mouse/keyboard alone, and Cohen et al. (1998) found that using vocal input with a pen increased the interaction speed of a
military application 8.7 fold. Based on this research, a multimodal approach should be considered, which is discussed further in Chapter 8.

3.5 Conclusion

When designing an application, it is important to consider ability-based design (Wobbrock et al., 2011), especially when designing the application for users with different capabilities, such as motorically challenged users. In order to focus on what users can do as opposed to what users cannot do, in the case of motorically challenged users, voice is a popular alternative. Voice has been used in both PBV and interface interaction (Begel, 2005; Begel & Graham, 2005; Begel & Graham, 2006; Dai et al., 2004; Désilets et al., 2006; Harada et al., 2009; Hubbell et al., 2006; Raman, 1996; Shaik et al., 2003). While not all PBV solutions were as successful as others (i.e., Hubbell et al. (2006) required too much typing and Begel and Graham (2006) found that speed and accuracy were issues that frustrated programmers), the overall consensus from user studies was that PBV is a viable alternative to the mouse/keyboard (Begel, 2005; Begel & Graham, 2005; Begel & Graham, 2006; Désilets et al., 2006; Shaik et al., 2003). Voice is also a suitable alternative for interface interaction whether it be input (Dai et al., 2004; Harada et al., 2009) or output (Raman, 1996). Vocal strain is a concern for PBV (Haxer et al., 2001; Oviatt, 2000); therefore, additional input modalities should be considered, particularly multimodal interaction (Alsuraihi & Rigas, 2007; Bolt, 1980; Cohen et al., 1998; Neto et al., 2008).
CHAPTER 4
MYNA: HANDS-FREE PROGRAMMING

Many existing IPEs (e.g., Scratch, Mindstorms, and Blockly) were not built with a Vocal User Interface (VUI) in mind; therefore, adding a VUI to these tools can be complicated. Despite the need for “out of the box” solutions to lower the usage barriers for those with impairments (Wobbrock et al., 2011), an “out of the box” voice recognition system (e.g., Dragon Naturally Speaking) will not integrate with these legacy applications without a bridge to connect the voice recognition system to the legacy application. This bridge needs to connect vocal commands to those the user sees on the screen. Moreover, the voice recognition system must be able to accommodate the behavior of the application, which is a far more difficult process. There is a significant difference in adding a “click” or “drag and drop” action than an action to “drag and drop” one block within another (it requires more than simple navigation). Furthermore, the VUI must allow the user to perform all necessary tasks (e.g., delete), which may require the creation of additional commands not native to the GUI due to the WIMP metaphor (i.e., the user would normally perform multiple steps to accomplish the task with a keyboard/mouse, but the VUI must have a single, vocal command that initiates the same behavior). Because the commands are not native to the application, additional code must be written to handle the appropriate behavior. This could be seen as an advantage for the vocal user (i.e., state one vocal command, versus perform three steps with a keyboard/mouse).
This chapter describes early Myna, the original prototype, in Section 4.1 followed by the summary of a user study to evaluate what functions users perform in Scratch with the mouse/keyboard to determine what Myna is missing (Section 4.2). Section 4.3 lists four mapping challenges encountered when creating a VUI for the Scratch GUI and explains how these challenges were overcome. Section 4.4 outlines Myna requirements, limitations, and user assumptions followed by a conclusion in Section 4.5.

4.1 Overview of Early Myna

The PBV research summarized in Chapter 3 indicates that voice is a viable input modality for providing accessibility for those users with motor impairments. A tool affectionately named Myna, which is a species of bird that are well known for their imitative skills, was created by previous students of Dr. Gray at the University of Alabama Birmingham through support from a Google Research Award (Rudraraju et al., 2011). A prototype was created and populated with a small vocabulary to perform basic functionality (Rudraraju et al., 2011). Myna has been extended in this dissertation as a Java program that runs parallel to Scratch (chosen due to its large user base as mentioned in Chapter 1) utilizing Sphinx (CMU Sphinx, 2015) as a voice recognition system to allow for further portability. When Myna is started and Scratch is opened, the user can begin creating a program within Scratch solely through voice. The grammar for the user is relatively simple due to the verbal commands matching the commands on the screen with the exception of action commands (e.g., “drag and drop,” “drop after,” “delete”). The grammar has been kept as small as possible to maintain Papert’s philosophy of “low floor” (Papert, 1980). The extension of Myna was supported by NSF grant IIS-1117940.

---

7 Myna project page: http://myna.cs.ua.edu
4.1.1 Design of Early Myna

Another design goal was to use an architecture that is robust enough to accommodate future changes in Scratch’s user interface (e.g., if future versions of Scratch move menu items and widgets to different parts of the Scratch screen). To assist in separating the dependencies between the user interface and the core functionality, the Model View Controller (MVC) (Gamma et al., 2000) design pattern was applied because it is commonly used to maintain large and complex data sets involved in developing user interfaces. In the following subsections, a component is defined as a clickable area within the Scratch window, which generates an action in Scratch.

The model represents the category, state and low-level behavior of the component blocks as illustrated in Figure 4.1. The Component Mappings consists of all of the property files used by Myna (see Appendix A, Myna Documentation, for more information about property files). Each property file contains xy-coordinate information about a specific component (see Listing 4.1 for

![Myna Architecture Diagram](image)

**Figure 4.1.** Myna Architecture (Rudraraju et al., 2011; Wagner et al., 2012).
an example of the “add” property file). The most important fields within the property file are: name, initialLocationX, initialLocationY, length, height, numParams, parameterLength1, and parameterLength2. Name is obvious, but this must match the word used in the grammar to refer to the particular block. InitialLocationX and initialLocationY refer to the xy-coordinate of the block on the Scratch screen. It can refer to any xy-coordinate where the mouse would be able to click on the component to drag it onto the screen. Length and height are used to calculate block placement in the block editor within Scratch. NumParams is necessary to allow for the editing of parameter values as is parameterLength1 and parameterLength2, which document the distance of the parameter from initialLocationX. Described further in Section 4.3, NumParams is used in the algorithm that updates subsequent parameter locations upon the editing of a parameter.

When a block is added to the block editor in Scratch, details regarding the block are inserted into two data structures representing state storage. The component and its data are stored in a Vector. Each component is then mapped to a number and stored in a HashMap. These numbers are required for ease of navigation (discussed further in Section 4.1.2, Transparent Frame Navigation). The number is displayed in a small Java Frame placed next to each block.

Listing 4.1. Property file for the “add” block.

```
name = add
nickname = o1
parentName = operators
initialLocationX = 36
initialLocationY = 196
isInFocus = true
length = 47
height = 20
numParams = 2
parameterLength1 = -18
parameterLength2 = 13
```
where everything but the label displaying the number is transparent. This Frame overlays Scratch and allows the user to manipulate a block in the editor by referencing the number. The Sprite State also keeps track of the drop location on the screen, which is how Myna knows where to drop a block in order for the sequence of blocks to connect.

As previously mentioned, a component is a clickable area within Scratch, which consists of three different types of components (see Figure 4.2 for a visual representation of these components):

1. Static components – these components are not movable (e.g., File, Edit, Scratch menus);
2. Movable blocks – blocks that can be dragged from their current location and placed into the block editor (e.g., add, when clicked, move steps); and
3. Movable block containers – movable blocks that can contain other movable blocks (e.g., if, repeat, forever).

Figure 4.2. Scratch with the three types of components identified.
The property files for each component are organized into one of these three folders. This allows Myna to load the correct property fields given a specific component type. For example, static components only have the fields: name, nickname, parentName, initialLocationX, and initialLocationY. Myna does not need to know the height and length for static components because they are not dragged onto the screen. Movable block containers have additional fields within the property files because Myna must know where within the container to insert other movable blocks.

The view is a representation of the model visible to the user. The Scratch user interface serves as the View since the actions performed by Myna are visible on the screen through the Scratch interface. Additionally, the numbers mapped to the components in sprite state storage are also visible on the screen via transparent windows. A small Java label is placed on the screen with the remainder of the Java window being transparent. These numbers are hereafter referred to as transparent frames.

The controller performs the required action in the Scratch window based on the user’s vocal commands. Four components make up the controller:

1. Speech Recognizer: The speech recognizer is used to recognize the vocal commands given by the user. Early Myna used Cloud Garden, a third-party implementation of JSAPI (Java Speech API), for speech recognition; however, Myna now uses Sphinx from Carnegie Mellon University (CMU Sphinx, 2015) since it reduces the user setup of Myna. Cloud Garden required an additional install (TalkingJava SDK), which could prove to be difficult if the Java and Classpath environment variables were not set up properly. Sphinx is a .jar file that is now packaged within the Myna .jar file resulting in one file for the user to download and execute.
2. Grammar: The Grammar is a Java Speech Grammar File that contains all vocal commands implemented in Myna. All of the components are included in the Grammar in addition to action commands such as “drag and drop,” “delete,” and “pause.”

3. Command Executor: All vocal commands are processed into an action and a component (if there is a component, some commands, e.g., “pause,” do not have an associated component). The Command Executor contains all of the methods associated to the actions possible within Myna and invokes the Java Robot class to perform the required mouse/keyboard behavior.

Listing 4.2. “Drag and drop” code.

```java
public void dragAndDrop(AbstScratchComponent comp) {
    int currentX = comp.getX();
    int currentY = comp.getY();

    ScriptsState scriptsState = AppState.getCurrentScriptsState();

    int tempCurrentDropPointX = scriptsState.getDropX();
    int tempCurrentDropPointY = scriptsState.getDropY();

    robot.mouseMove(currentX, currentY);
    robotMousePress();

    robot.mouseMove(tempCurrentDropPointX, tempCurrentDropPointY);

    //change components current location
    comp.setX(tempCurrentDropPointX);
    comp.setY(tempCurrentDropPointY);

    //add the component at the end of the Sprite State Vector
    AppState.getCurrentScriptsState().getMovableComponents().add(comp);

    //update current drop point
    int compHeight = comp.getHeight();
    this.moveCurrentDropPoint(compHeight);
    robotMouseRelease();
}
```
4. Java Robot: The Java Robot class is part of the Java.awt library (Oracle, 2015). It generates mouse and keyboard events (e.g., click, right-click, type a letter) allowing users to programmatically control the screen. For example, to implement “drag and drop,” the user issues the following command, “drag and drop move steps.” Myna then gets the location of the “move steps” block from the corresponding property file, and then, Myna invokes the Java Robot to move to the coordinates stated in the property file. Myna tells the Java Robot to press, which causes the typical mouse behavior of clicking to occur. The “move steps” block has now been “grabbed” by the mouse (without the user having to physically touch the mouse). Myna then determines the drop location and commands the Java Robot to move to this new coordinate while still holding onto the “move steps” block. Once the appropriate drop location is reached, Myna commands the Java Robot to release the mouse thereby dropping the “move steps” block. The “drag and drop” code can be found in Listing 4.2.

4.1.2 Functionalities of Early Myna

When the user creates a program in Scratch using Myna, the mouse moves without the need for any physical activity from the user. As mentioned above, this is due to the Java Robot class executing the mouse and keyboard events translated by the speech recognition engine. Myna allows the user to have full control over the interface with just his/her voice. The following is an example of how Myna maps vocal commands to actions (refer to Figure 4.3) (Rudraraju et al., 2011; Wagner et al., 2012; Wagner 2013):

1. User gives a vocal command.

2. The input command is identified by the Speech Recognizer, parsed, and checked against the Grammar file.
3. If the command is present in the grammar file, an appropriate action is invoked in the Command Executor.

4. The Command Executor obtains the current mappings of the component.

5. The type of block is validated against the Component Hierarchy ensuring all required fields are present.

6. The Command Executor calls into the Java Robot class to perform the corresponding mouse/keyboard action.

Myna allows three types of navigation (Rudraraju et al., 2011; Wagner et al., 2012):

1. Drag and Drop Navigation: This mimics the idea of clicking on an object, dragging it to another location, and dropping it. The user will say, “drag and drop” followed by the movable component block to add to the program, and the block will be placed after the last block in the program.

2. Continuous Navigation: The user drives the cursor by stating, “move right” and “keep moving,” which contradicts research indicating that target-based commands result in less

Figure 4.3. Myna Workflow – mapping of voice commands to actions in Myna.
errors and are easier for users than direction-based commands (Harada et al., 2009; Sears et al., 2003); therefore, this might be an area for future study.

3. **Transparent Frame Navigation**: The transparent frames allow small numbers to be placed next to commands within the program (see Figure 4.4). The user will state “drag” followed by the command on the desired block, and upon determining where the user wishes to place the block, he/she will state one of three macro commands (“drop before,” “drop in,” or “drop after”) and the number from the associated label. Furthermore, if a user wishes to delete a block, the user will say, “delete” followed by the number next to the block to delete. This allows the user to identify the target block simply. If the user had to name the block rather than use a number, conflicts could arise. For example, if there were multiple “move steps” blocks in a sequence, how would the user be able to discern between them? This is the primary motivation behind the use of transparent frames.

4.1.3 **Limitations of Early Myna**

Early Myna was only a prototype; therefore, it did not include all of the necessary commands for a user to interact with Scratch. The original grammar consisted of 14 commands and 38 components. The grammar in the current version of Myna consists of 36 commands and
162 components. This may seem like a large grammar, but the components in the grammar have a one-to-one relationship with the components within Scratch, and the number of commands has been kept as minimal as possible.

Early Myna was difficult to install. It required third-party installations in order to download and install Cloud Garden (speech recognition engine) and any dependents. A graduate student at the University of Alabama (UA) changed the speech recognition engine to Sphinx from CMU (CMU Sphinx, 2015), which is written in Java. This allowed the speech recognition engine to be packaged within Myna resulting in the end-user to only need to open one .jar file in order to begin using Myna. Additionally, this graduate student added in the features to load a saved Myna program. In Early Myna, a user could not save his/her work. The Scratch program would be saved, but the Myna program would not. If the user chose to perform additional work on the Scratch program, he/she would be unable to do so. There now exists a Save function in Myna so that both the Myna and the Scratch programs are saved.

There were several other limitations within early Myna such as the ability to pause the speech recognition engine or edit parameters. More issues emerged after conducting a Scratch user study, discussed in Section 4.2, and these issues are explained in detail in Section 4.3.

4.2 Learning from Scratch

During the summer of 2013, UA offered multiple workshops to both teachers and students. A three-day teacher workshop introduced teachers to fun CS tools to use in their classrooms. A week-long middle school camp taught students CS fundamentals using applications aimed at inspiring young students to begin studying CS. Both the teacher and student workshops had similar content and activities, lending to a common observational study
about how users behave when writing programs in Scratch. This study is described in Section 4.2.1, and implications for Myna are discussed in Section 4.2.2.

4.2.1. Observations of Scratch

To evaluate how users of varying age and experience levels write programs in Scratch, both teachers and middle school students were included in the study. The teachers attended a three day workshop spending 1.5 days working in Scratch; the students attended a week long introduction to CS where two days were spent working in Scratch. During initial instruction, instructor-led activities, and independent activities, the usage behavior of students and teachers was observed, while noting which blocks were used and in what order.

During the observation portion of the study, three primary behaviors were noted. Below is a list of these three behaviors including an explanation of why these behaviors are important.

1. Did the participant drag blocks onto the screen (possibly blocks not required for the assigned program) without immediately connecting them? When teaching Scratch, only the next required block is dragged onto the screen. Did participants use a different approach?

2. If the participant dragged additional blocks onto the screen, were these blocks eventually used, deleted, or left on the screen? Myna does not allow a user to place a block randomly on the screen. Should this functionality be added?

3. Did the participant start his/her program with an event block as demonstrated by the instructor? In Scratch, every program script must begin with an event, or that script will not be executed. Myna was not designed for the first block in the script to be added later. Although “drop before” could work, Myna is inconsistent with its ability to add a block to the beginning of a script.
In addition to these behaviors, it was noted whether a participant had previous experience using Scratch. How these behaviors changed over time as instruction continued and the complexity of the programs increased was also observed.

The teachers were the first group to be observed. Scratch was only taught the first 1.5 days of the teacher workshop. The morning began with an instructor-led session of creating a square in Scratch, which the teachers had the opportunity to modify to create a different shape if desired. Teachers were then guided through the creation of a grade program that averaged three grades and were then given the opportunity to alter the program to average more than three grades or to make other alterations as desired. Finally, teachers were guided through the process of creating a maze game that required more complex problem solving by needing cursor control, stage background alterations, and collision detection. Table 4.1 presents a summary of the data collected through observation for 20 teachers (two had previous experience in using Scratch). “Part” refers to when the programs increased in complexity throughout the day (these parts are split into days for the middle school camp). Of those who dragged additional blocks onto the screen during Part 1, three deleted the blocks upon discovering they were not needed for the given problem, and only one participant left the block on the workspace. During Part 2, that same participant deleted the extra blocks.

The middle school students were the next group to be observed. The middle school camp consisted of two days of Scratch. The content of the first day was similar to the content taught during Part 1 of the teacher workshop. Students were taught how to draw a square by following

<table>
<thead>
<tr>
<th>Part</th>
<th>Additional Blocks</th>
<th>Not Event Block First</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
along with the instructor. The instructor then guided the students through the process of creating a program that would draw any number-sided shape (creating a generic program), and then, students were asked to modify the program in any way he/she desired. The next program taught was the same grade program the teachers created, and students were allowed to modify it. The students were then taught to manipulate costumes with a walking character on the screen. On Day 2, students were guided through the same maze program taught to the teachers, and finally, each student was given the opportunity to create his/her own program to present the morning of Day 3 (which was the Parent Showcase event where students demonstrated their creations to their family). Table 4.2 presents a summary of the behavior data for 28 students (four had previous experience in using Scratch) during the observation portion of the study. Of the ten students who dragged unnecessary blocks into the editor on the first day, four used the extra blocks, four left the blocks in the editor (unused), one deleted the blocks, and one used the extra block as a fast way to change his program to perform a different task. On Day 2, three of the four deleted the excess blocks, and one left the block in the editor (unused).

Throughout the observation period, participants who dragged unnecessary blocks onto the screen were asked why he/she chose to do so. The most common response was interesting: the participants felt it helped him/her solve the assigned problem. When asked why a participant chose not to delete excess blocks, one participant stated that he did not make time to do so yet and another participant stated that he left it as an optional modification. The latter response was in reference to a shape-drawing program. If the participant wanted to draw one type of shape, he

<table>
<thead>
<tr>
<th>Day</th>
<th>Additional Blocks</th>
<th>Not Event Block First</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
could easily connect the appropriate event block to the extra script, and the desired effect would occur. A teacher mentioned that he liked to leave extra blocks on the screen in the situation the blocks were needed for something later. Another student commented that rearranging the blocks helped her think through the problem and allowed for increased success.

4.2.2. Implications for Myna

As described above, participants dragged extra blocks into the editor to aid in solving the assigned problem. There are approximately 125 commands available in Scratch (2015); thus, participants were shrinking the 125 commands down to a smaller subset by dragging the extra blocks into the editor and then creating an even smaller subset by achieving the necessary code segment. Over time, however, fewer participants were utilizing the extra blocks. The participants were able to skip the middle step and were able to take the 125 commands and create the final subset necessary for the code segment. This change tells us that given more time to work in Scratch, the participants were able to solve more complex problems with more efficiency.

There was also an opportunity to practice differentiated instruction with the students. Throughout the Scratch portion of the middle school camp, the students with previous experience were given extra challenges that were new. All four students who had previous experience dragged unnecessary blocks (or blocks out of order) into the editor in order to think through the problems. The primary reasoning for this was to pull blocks that might help solve the problem. These users also either used or deleted the extra blocks; they did not leave them in the editor.

Overall, during Part 1 of the Scratch lesson, 31 percent of the users (students and teachers) dragged extra blocks onto the screen. While this percentage decreased to 10 during the second portion of Scratch, the students with previous Scratch experience felt it was a helpful way to solve a given problem. Myna currently does not have this functionality, but it is a feature that
may need to be considered and added to the design; however, given the way blocks are accessed via voice, this may have to remain a limitation of Myna. Myna stores a representation of the screen internally via vectors and hashmaps. This representation depends on blocks being added to the editor through the use of the pre-defined commands: “drag and drop,” “drop after,” “drop before,” and “drop in.” If a user wanted to add random blocks onto the screen, the user could do so using continuous navigation (directing the mouse vocally); however, Myna would not be aware that the block was placed in the editor. Thus, the user would be responsible for manipulating this block. For example, the user would not be able to use the “delete” command to remove it. The user would have to again use continuous navigation to remove the block from the editor.

Regarding the 15 percent of users not placing a control block first in the program during Part 1 of the Scratch lesson, it was good to see this percentage drop to four during the Part 2, but users should be able to delete the initial block and replace it with a different control block, if desired. 19% (nine out of the 48, all students) of the users spent a significant amount of time playing with different sprites and manipulating the stage. The fact that these users were all students demonstrates that younger users enjoy manipulating the stage. This is a feature that needs to be added to Myna.

4.3 GUI Mapping Challenges

While creating Myna, we discovered an assortment of needed features (e.g., pausing speech recognition, horizontal parameter expansion, dynamic GUI changes, and deleting blocks) that presented new challenges in mapping a GUI to a VUI due to losing the context of the xy-coordinate (i.e., the location was moved because of another action) or because the command was
not native to Scratch. These four challenges and their corresponding solutions are described below:

1. **Parameters – horizontal expansion**: Various commands (e.g., “go to x y”) in Scratch require multiple parameters (Figure 4.5a). Depending on the length of the information entered in the first parameter slot, the xy-coordinate of the latter parameter slot(s) will change (Figure 4.5b). Since Myna must maintain a representation of where all components are located on the screen, changing the width of a parameter slot affects Myna’s understanding of where subsequent parameter slots are located. For example, the property file of a component tracks the distance from the “grab point” to each parameter slot. If there are multiple parameter slots in a component, and the first parameter slot is edited, the distance from the “grab point” to the subsequent parameter slots will increase. Thus, the component’s information must be updated in the Myna representation, or when a user attempts to edit a subsequent parameter slot, Myna will direct the mouse to the incorrect location.

   **Solution**: The distance from the beginning of the command block to each parameter slot is stored in a property file. As the user enters information vocally, Myna captures the information to measure the length and edits the corresponding component stored in the sprite state. For example, if the user submits the number “100” in the first parameter of the block in Figure 4.5b, Myna takes a constant (the width of one character – 8 pixels), multiplies it by three (the length of the string entered), and calculates the new distance.
from the latter parameter slot to the beginning of the command block. Myna uses the NumParams property to determine how many parameter slots must be updated (i.e., if there is only one parameter slot, no update is necessary, but if there are three parameter slots, then the two subsequent parameters must be updated). If a variable is inserted in a parameter slot (Figure 4.5c), a similar process is used. The variable’s name is stored when the user creates the variable; based on the name, Myna calculates the length of the variable name and adds the edges surrounding the string in the variable block (8 pixels * number of characters in the name + 7 pixels for the edges). The width of the edges of the variable block is consistent for all variables.

The algorithm to edit a parameter consists of two methods. The first method, “setParameterForComponents,” (see Listing 4.3) moves the mouse to the proper position depending on which parameter the user wishes to edit. The second method, “editParameter,” (see Listing 4.4) is invoked anytime the user issues the “number” or “letter” command. As an example, if the user were to edit the second parameter of the fifth block (see Figure 4.6), the user would say, “edit two at five.” This command will invoke the first method (“setParameterForComponents”, Listing 4.3). The user would then say how to edit the parameter, “number 1, number 2.” This would invoke the second method (“editParameter”, Listing 4.4) and would change the parameter’s value to 12.
**Listing 4.3.** Algorithm identifying which parameter to edit.

```java
void setParameterForComponents(MovableComponent target, int paramNumber) {
  Display HelpWindow explaining how to edit parameters
  class variable: editedComponent = target
  if(paramNumber equals 1)
    Move the mouse to (target.x + parameterLength1, target.y)
    //parameterLength1 comes from the property file of target
  else if(paramNumber equals 2)
    Move the mouse to (target.x + parameterLength2, target.y)
    //parameterLength2 comes from the property file of target
  else if(paramNumber equals 3)
    Move the mouse to (target.x + parameterLength3, target.y)
    //parameterLength3 comes from the property file of target
  Mouse click
  Mouse release
}
```

**Listing 4.4.** Algorithm to change the value of a parameter.

```java
void editParameter(int value, String val) {
  Delete existing value
  Enter value from user
  Store value in class variable temp
  if(editing a parameter)
  {
    Get number of parameters from property file: numParams
    if(length of temp greater than 1)
    {
      if(numParams greater than 2) //3 parameter slots in block
      {
        if(parameterNumber equals 1) //edit first parameter
        {
          Increase parameter length 2 by 8 pixels
          Increase parameter length 3 by 8 pixels
        }
        if(parameterNumber equals 2) //edit second parameter
          Increase parameter length 3 by 8 pixels
      }
      else if(numParams greater than 1) //2 parameter slots in block
      {
        if(parameterNumber equals 1) //edit first parameter
          Increase parameter length 2 by 8 pixels
      }
    }
  }
  else //implies naming a variable rather than editing a parameter
  Concatenate val to Variable name
}```
2. **Delete**: Scratch does not provide a “delete” command. Instead, the user drags the block to be deleted from the code editor to the command palette on the left side of the screen or right-clicks on a block and chooses “delete.” If the desired block is in the middle of a code segment (Figure 4.7), the user must separate the desired block from the segment, drag the desired block to the command palette, and reconnect the remaining blocks in the code editor. If the user were to simply drag the block to be deleted to the command palette (or right-clicked on the block to be deleted and selected “delete”) while the subsequent blocks were still connected, all of the subsequent blocks would also be deleted.

**Solution**: The solution, while more difficult to implement within Myna, is easier on the end-user than the mouse/keyboard method for deletion. The user will state, “delete” followed by the number of the block to be deleted (Figure 4.8). Using the Java Robot class and information regarding the location of the blocks on the screen, Myna separates the blocks within the code segment, if necessary, and drags the desired block to the
Listing 4.5. Delete algorithm.

```plaintext
void delete(Component toBeDeleted) {
    Store x and y coordinates of block toBeDeleted

    //Determine if toBeDeleted is the last element, if so remove toBeDeleted from
    //the end of the vector, else remove from between
    if(toBeDeleted is not the last element)
    {
        //Separate blocks
        Get the x and y coordinates of the element after toBeDeleted
        Store x and y
        Move the mouse to this location and click
        Drag these blocks to the right 10 pixels and down 50 pixels
        Release blocks

        //Delete designated block
        Move mouse to (toBeDeletedX, toBeDeletedY)
        Click block
        Drag block to left-hand side of screen and release

        //Get location of block to move back
        Move mouse to location of block to move back (this was stored earlier)
        Click block
        Drag block to original location of toBeDeleted (stored in the beginning)
        Release block
        Remove toBeDeleted from vector
        Update transparent frames

        //Update drop point height
        Calculate block height * -1 to get distance to subtract
        Move current drop point distance calculated above
    }
    else
    {
        //Delete at the end
        Get index of toBeDeleted
        Move mouse to block to be deleted
        Click block
        Drag to left-hand side of screen and release
        Remove toBeDeleted from vector
        Update transparent frames

        //Change drop point height
        Calculate block height * -1 to get distance to subtract
        Move current drop point distance calculated above
    }
}
```
command palette, mimicking what the user would do with the mouse. After deleting the block, the number labels are re-generated to avoid leaving an empty label on the screen.

The pseudocode for the delete method can be found in Listing 4.5. When the user invokes the delete command by stating, “delete” followed by the number of the block to delete (i.e., “delete 5” in Figure 4.8a), the delete method will be called and passed the component at position five in the list.

3. **Dynamic GUI changes**: Scratch allows users to create custom variables and lists. The Variables menu in Scratch begins with two buttons (Figure 4.9a), but when the user adds a variable or a list, the variable or list is added to the menu along with associated commands (Figure 4.9b). The user can add as many variables or lists as desired; however, with each additional variable or list, the xy-location of the items listed below it will shift (Figure 4.9c). Also, the user is responsible for naming new variables and lists, which must be stored for future use.

*Solution*: Myna is unable to edit property files during runtime; therefore, all information is captured in data structures stored in state files. Regarding the naming issue (i.e., the user declares the name of the variable upon creating it), when the user creates a new
variable or list, Myna creates a new Variable object and adds one letter at a time (as the user verbally types the name in) to the variable’s name. The xy-coordinate of this variable is grabbed from the ‘New Variable’ property file. This property file serves as a dummy variable to give the initial xy-location of the first variable created. Myna must track the number of custom variables and lists created in order to calculate the xy-coordinate of the next new item (Figure 4.9c). The variables are stored in an ArrayList in the AppState file. For each new variable added, 20 pixels (the height of the variable block plus the distance between variables) are added to the y-value of the xy-location of all components below the variable being added. While lists have not yet been implemented, they would be implemented identically to variables.

4. **Pause**: “Pause” is a non-native command primarily because there is no need for it within Scratch. However, in Myna the speech recognition engine is always in the listening state.
and interpreting utterances. If someone were to enter the room and begin speaking to the user, the speech recognition engine would receive the conversation and begin trying to perform actions based on the utterances collected; therefore, the user requires the ability to “pause” the application and “resume” it as needed.

Solution: When the user says, “pause,” a dialog window will appear informing the user that Myna has been paused. A new thread will start allowing the speech recognition to continue monitoring; however, only the word “resume” is a valid command. After the user says, “resume,” Myna will return to its normal state.

4.4 Myna Requirements, Limitations, and User Assumptions

Myna is written in Java as is the Sphinx speech recognition engine; therefore, Myna is platform-independent, and any computer able to run Scratch v1.4 should have the capability of running Myna. A directional microphone (roughly $30) is recommended because it helps eliminate ambient noise. Myna does not work well in a noisy environment; and because Myna is a VUI, it requires the user to speak, implying that a classroom may not be the best place to use Myna, unfortunately.

Although Myna now has the ability to edit parameters, the editing is done one letter/number at a time. This could become very frustrating to the user in the event that the user wants to enter a large phrase (e.g., the user can enter sentences that are displayed above the sprite’s heads in clouds or bubbles). This is clearly a limitation that should be addressed. Also, if the parameter consists of a dropdown list, it cannot currently be edited. There is currently no ability to right-click in Myna. Although having the option to right-click is a nice feature to have, it is not necessary, and in fact, the right-click menus in Scratch are very restrictive. As previously
mentioned, only one variable can be added in the current version of Myna, and creating lists is not yet functional.

The major assumption for Myna is that users have the ability to speak articulately. Users should also have the cognitive ability to learn to program in Scratch (equivalent of a typically developed eight year old). Moreover, it is helpful if the user has a general interest in learning to program.

4.5 Conclusion

Although the current version of Myna still possesses limitations, many improvements and modifications were made from the pre-existing prototype. Some of these limitations leave possibilities for future work. It may be that the cursor control necessary for some of these limitations (e.g., selecting a value from a dropdown list) would be more suitable using a different input modality (discussed further in Chapter 8, Future Work). The challenges reviewed in this Chapter should highlight the difficulty in creating a VUI for a GUI because although Scratch is a simple GUI, it has many different types of components all requiring different interactions.

The observation of Scratch users suggests that while there are some user actions that are not included in Myna, Myna does allow for the majority of user actions. The primary actions not provided for in Myna include dragging additional blocks (i.e., unnecessary blocks) onto the screen for later use and stage manipulation. The former action is not reasonable within Myna due to Myna having to control where each block is placed. For example, if an extra block were to be dragged onto the screen by Myna and placed in the bottom, right corner of the block editor, it might eventually get covered by the growing program, which could cause confusion for the user. As previously mentioned, both stage manipulation and the ability to delete/add the first block should be included in Myna.
CHAPTER 5

USER EVALUATIONS OF MYNA

To get a better understanding of the feasibility of Myna, three clients of the United Cerebral Palsy of Greater Birmingham (UCP) reviewed Myna. After the changes described in Chapter 3 were made, two studies were conducted: the first was a pilot study with five graduate Engineering students, and the second was a target user study with clients from UCP. Each study is described individually below. This chapter describes the UCP client review (Section 5.1), a pilot study (Section 5.2), and a target user study (Section 5.3). The data collected from the two studies is discussed in Section 5.4 followed by an ability-based design evaluation in Section 5.5 and lessons learned in Section 5.6.

5.1 UCP Client Review Evaluation and Discussion

In order to garner feedback from target users regarding the viability of Myna, a preliminary review was conducted at UCP. A UCP Employment Specialist arranged a meeting with three individuals each of whom experienced motor impairments.

The first client faces extreme motor impairments; he is unable to move any part of his body other than his head, and he has limited control over that movement. He communicates via a laptop. He first types what he wants to say using a device he wears from his chin, and the laptop speaks for him. The program on the laptop contains menus for various categories of phrases (e.g., greetings, common phrases) in order to improve the efficiency of the process. I demonstrated Myna to him, and then he wrote a short program by typing the commands into his
laptop and his laptop then spoke to my laptop. He was able to program successfully, and he really enjoyed working in Scratch. He thought it would be helpful for him if I could import the grammar into his program. Then, he would not have to type as much to achieve the same goal. Clearly, a different type of interface would work even better for him enabling Myna to exist solely on his laptop, but discovering that automated speech works with Myna was helpful.

The second individual also faces motor impairments; however, they are not as severe as the first individual. Although the second individual can speak, her articulation is not clear enough for the Sphinx voice recognition engine. Thus, she was unable to test Myna herself, but she did observe Myna in action. She suggested adding other input modalities such as eye tracking, particularly since she used eye tracking to enter information in a database in her daily job.

The third individual is a quadriplegic due to an accident at work. His speech was clear, and he was able to test Myna. He echoed the first two individuals’ thoughts that Myna was an interesting tool and held value for the motorically impaired population. He was also excited about the idea of being able to learn how to program as he felt a programming type job would be appropriate for him in the future.

The three design reviews above demonstrated that Myna did have viability, and it grabbed users’ interest. I also learned that additional input modalities would be important in the future (discussed in Chapter 8, Future Work), and that even if a person’s articulation is clear, he or she still needs to practice using Myna in order to perfect the articulation of Myna’s grammar. Finally, a downloadable version of Myna’s grammar would be helpful to users possessing various accessibility devices.
5.2 Pilot Study Evaluation and Discussion

Although it is important for Myna to function as well as the corresponding Scratch GUI, the goal is not to compare the two to each other, but to provide similar functionality requiring the participants in the pilot study to use both tools. The pilot study participants consisted of five male, Engineering students from the University of Alabama. Four of the study participants had completed a graduate degree, and the fifth had completed some graduate course work. Three of the five were native English speakers, and the other two students were non-native English speakers.

The type of study performed was a one-group post-test only approach (see Figure 5.1). The target population cannot currently utilize Scratch because of the dexterity required by the Windows Icon Mouse Pointer (WIMP) metaphor; therefore, a control group is not reasonable. The initial pilot study was intended to evaluate the study design and instruments used; however, the pilot study also provided an opportunity to evaluate the current state of Myna.

The following hypotheses were evaluated in the pilot study:

$H_0$: Myna is not a suitable alternative to the mouse/keyboard.

$H_1$: The time difference between the mouse/keyboard and Myna is not significant.

$H_2$: Myna is easy to learn: The number of errors (e.g., saying the incorrect Myna command) occurs less with each use.

$H_3$: The implemented solutions for Delete, Pause, and Parameters are successful.

![Diagram of Pilot Study Design](image)
5.2.1. Pilot Study Testing Procedures

The participants evaluated Myna using a laptop and microphone, where each participant was given a script with three programs to write (see Appendix B for the Pilot Study Testing Instructions). The programs were designed to be very simple, but they each test various commands such as: “drag and drop,” “drop after,” “pause,” “delete,” “set property at” (parameters), “help,” and menu navigation. The simplicity of the programs allowed the participants to focus on testing the “pause,” “delete,” and “set property at” (parameters) commands.

During testing of the Pilot Study, the participants created each program in Scratch using mouse/keyboard input first. The reasoning behind having the pilot participants start by using the mouse/keyboard method was twofold:

1. To time how long the participants took to complete the program using both mouse/keyboard and voice to determine whether voice took a comparable amount of time; and

2. To allow the participants to obtain a brief understanding of how Scratch works since the goal is to evaluate Myna’s usability rather than Scratch’s usability.

Next, the pilot participants recreated each program using Myna (voice only). Each participant’s experience using both input modalities was observed and documented. Finally, each participant completed an experience survey regarding how he/she felt about using Myna.

5.2.2. Data Collection

Two data collection methodologies were utilized: observation (Appendix C contains the Pilot Study Observation Instrument) and survey (Appendix D contains the Myna Experience Survey). No data was captured electronically. During the observation, errors were recorded: an
error could be the fault of the user (e.g., the user says the incorrect command) or the fault of Myna (e.g., speech recognition error or xy-mapping issue). The amount of time to complete each task was documented to determine if completing the task using the VUI took longer than when using the GUI. Furthermore, for any questions on the survey that appeared to contradict observed behavior (e.g., the participant appeared frustrated but commented otherwise on the survey), the participant was asked for an explanation, which was also recorded.

The survey asked more specific questions regarding the user’s experience during the experiment (Appendix E contains the Myna Satisfaction Survey results from the Pilot Study). The questions asked the participants about their opinion of using specific features of Myna (such as the solutions presented in the Mapping section); these included: if the commands were relevant and performed the expected actions, if help or error messages were useful, the user’s overall impression of Myna, and basic demographic information.

5.2.3. Pilot Study Results

The results from the pilot observation are presented in Table 5.1. Survey results are summarized throughout the Analysis section (Section 5.2.4) and the Discussion section (Section 5.4).

As can be seen in Table 5.1, there are four types of errors that can occur. Errors of Type A and D relate directly to Myna’s performance; errors of Type B and C, while of interest, are not directly related to Myna’s performance. To evaluate the ease of learning Myna, we focus on Type A errors, which occur when a user forgets one of the Myna commands (e.g., “drag and drop,” “set property at”) and causes an undesired action in Scratch. Type B errors occur when the user misreads or misunderstands one of the Scratch blocks (e.g., user says “think for seconds”
Table 5.1. Summary of observation data.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type A - Participant stated incorrect vocal command</td>
<td>0.8</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>(Average count)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type B - Participant stated incorrect Scratch command</td>
<td>0.6</td>
<td>0</td>
<td>0.4</td>
</tr>
<tr>
<td>(Average count)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type C - Speech recognition was inaccurate</td>
<td>4.8</td>
<td>2.4</td>
<td>1.8</td>
</tr>
<tr>
<td>(Average count)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type D - Myna placed the block in the incorrect</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>location (Average count)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to complete with Scratch (Average seconds)</td>
<td>114.2</td>
<td>83.2</td>
<td>88.6</td>
</tr>
<tr>
<td>Time to complete with Myna (Average seconds)</td>
<td>98.6</td>
<td>113.6</td>
<td>112.8</td>
</tr>
</tbody>
</table>

instead of “think”). This erroneous command was not what the program script stated; thus, it is a type of logic error rather than an error reflecting poorly on Myna’s functionality. Type C errors are due to speech recognition issues and are not a function of Myna’s performance. Although speech recognition errors give the appearance of being a Myna issue, our work is more focused on the viability of a VUI, not the evaluation of speech recognition. To avoid these errors, additional training must occur between the user and the speech recognition engine. Type D errors are Myna functionality issues, and need to be corrected before further user testing.

5.2.4. Pilot Study Data Analysis

In order to show that Myna is a suitable alternative to mouse/keyboard interaction, we must determine the degree to which the null hypotheses can be rejected. To do so, the remaining hypotheses must be supported by the observations and survey data collected. In order to support $H_1$, a paired t-test was performed comparing the time to complete the programs with mouse/keyboard and the time to complete the programs with Myna (see Figure 5.2). The result of the t-test is 0.323347 with an alpha of 0.05, which supports $H_1$. 
Figure 5.2. Time to complete using mouse/keyboard vs. Myna (U1-P1 = User 1 – Program 1).

Figure 5.3. Sum of errors per program.

For H₂, both quantitative and qualitative data was analyzed. First, the total number of errors due to a participant stating the incorrect Myna command (Type A error) were totaled and charted (Figure 5.3). Then, the qualitative data provided by the participants was analyzed, and the median of the participants’ responses was calculated. On all three questions, the median was four (“Somewhat easy”). Based on the number of errors decreasing dramatically from programs one and two to program three, in combination with the qualitative data provided by the participants, H₂ is supported: Myna is easy to learn.
To determine if expected functionality has an impact on perceived ease of learning, a chi-squared test was performed between two survey questions:

1. I feel the vocal commands in Myna perform the action that I expect ("Always," “Most of the time,” “Half of the time,” “Some of the time,” or “Never”); and

2. I perceive the degree to which Myna is easy to learn as ("Very difficult," “Somewhat difficult,” “Neither difficult nor easy,” “Somewhat easy,” or “Very easy”).

The result was 0.513, which is greater than alpha = 0.05. We may not have a large enough sample to adequately perform this test; therefore, this requires further investigation in future studies.

The percentage of the number of errors relating to each feature was calculated to evaluate H3. Of the 14 total Myna errors (Type D), one was due to Parameters; none were caused by Delete or Pause. Thus, 7% of the errors were caused by the special features mentioned in the Mapping section. User feedback was positive for all but Parameters (the median was calculated and resulted in scores of 4s and 5s for all categories except for Parameters, which received a score of 3); therefore, H3 can neither be supported nor rejected.

Errors of type C, speech recognition errors, have the highest occurrence. Two of the five Pilot Study participants were non-native English speakers, and their accent might have resulted in more errors (see Table 5.2 for the average number of errors per non-native/native speaker). To validate this idea, we performed a non-equivalent variance, one-tailed t-test with a result of 0.061, which is greater than alpha = 0.05, but it is not statistically significant.

<table>
<thead>
<tr>
<th>Table 5.2. Average number of speech recognition errors per person.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors per person</td>
</tr>
<tr>
<td>Errors per native speaker</td>
</tr>
<tr>
<td>Errors per non-native speaker</td>
</tr>
</tbody>
</table>
Program Two seemed to have the fewest speech recognition errors for non-native English speakers but the most speech recognition errors for native English speakers (see Figure 5.4). This highlights that word pronunciation differs for each user and suggests a need for a custom grammar wizard, which would allow users to customize the grammar to better match his/her needs. One native English speaker experienced zero speech recognition errors throughout all three test programs, which illustrates the accuracy of the speech recognition is based on articulation and pronunciation.

5.3 Target User Evaluation and Discussion

The UCP Study was conducted during a six-week summer camp for young adults (ages 14-23) sponsored by UCP. Participants worked one-on-one for approximately 30 minutes once a week. We spent the first week of the camp getting to know the participants in order to make the participants feel more comfortable working with us. The type of study performed was a one-group post-test only approach. Initially, seven attendees of the camp elected to participate in the study; however, only two attendees were both able and willing to participate (reasons for this are described in the Threats to Validity subsection).
5.3.1. UCP Study Testing Procedures

Participants were introduced to Scratch during the second week of camp and given a brief tutorial on how Scratch worked by creating a simple program (the program drew a square on the screen, and then the students edited the code to draw additional shapes). Session one (during week three of camp) began the vocal testing portion of the study, which was performed on the same laptop as in the Pilot Study with a directional microphone. At first, the participants were given an exact script of the vocal commands to execute in addition to a list of all available vocal commands. The exact script was provided during the target study because some of the participants did not have the cognitive ability to fully understand how to write the program; however, the participant who completed four sessions was able to program without the script by the fourth session. This participant spent the final session building his own program. Participants executed the commands while being observed (see Appendix F for the Observation Instrument used). At the end of the study, the participants completed a brief satisfaction survey (see Appendix G) regarding their experience using Myna. Only one participant completed all four testing sessions; a second participant completed two testing sessions (as the other two sessions were spent working on the meter of his speech); and two other participants did not have the vocal ability to complete a full session.

5.3.2. UCP Study Results

Because all participants in this study had a lack of fine motor skills, we only evaluated their use of Myna (as opposed to both the mouse/keyboard and Myna) was evaluated. The participants were not timed in order to ensure each participant was comfortable and did not feel pressured. Therefore, the data collected during the observations was minimal and focused primarily on the same errors documented during the Pilot Study. The errors observed for
participant one are listed in Table 5.3. The errors observed for participant two are listed in Table 5.4.

5.3.3. UCP Study Data Analysis

Due to low levels of participation, there is very little data from the UCP Study to analyze. In Tables 5.3 and 5.4, we see few errors (one) of Type A and B across all four sessions. This was primarily due to the fact that we wrote out the vocal commands necessary to create the specific program. The participants only had to read the scripts when using Myna; however, Participant One was able to use Myna to create his own program by Session Four without Type A and Type B errors. Participant Two needed assistance with understanding the script because his reading level was not advanced enough to read the scripts.

Table 5.3. Error counts for Participant One.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Session 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type A - Participant stated incorrect vocal command</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type B - Participant stated incorrect Scratch command</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type C - Speech recognition was inaccurate</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type D - Myna placed the block in the incorrect location</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.4. Error counts for Participant Two.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Session 1 – Invalid</th>
<th>Session 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type A - Participant stated incorrect vocal command</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type B - Participant stated incorrect Scratch command</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type C - Speech recognition was inaccurate</td>
<td>5 – stopped session</td>
<td>2</td>
</tr>
<tr>
<td>Type D - Myna placed the block in the incorrect location</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Type C errors also decreased for both users from session to session, which represents Myna’s goal. The Type D errors that occurred were related to incorrect vocal commands and speech recognition errors.

5.4 User Testing Data Discussion

Regarding the Pilot Study, in the first program, Myna averaged 15.6 seconds less time than the mouse/keyboard; however, in programs two and three, Myna averaged 30.4 and 24.2 seconds longer than the mouse/keyboard for a total average increase of 13 seconds. In the second program, the increase in time might be due to the extra step of using “pause” (there is no pause function in Scratch; it is not necessary when using the mouse/keyboard). Some participants, however, took less time to complete the program in Myna. Based on the small increase in time on average and decrease in some circumstances combined with the t-test value of 0.32 (which is greater than the alpha), Myna may not increase productivity, but the increase in time is not significant. Again, the goal is not to replace the mouse/keyboard input modality, but to instead provide an alternative option for those who cannot use the mouse/keyboard.

The number of errors occurring on average is minimal with the exception of speech recognition errors. Speech recognition often requires some training for both the speech recognition engine and the user. The particular recognition engine used in Myna, Sphinx, does not require training, but the user should still practice articulating the pre-defined grammar and meter. The largest number of errors occurred for participants who are non-native English speakers. If the extreme outlier of participant U4 for program one is removed, the average number of speech recognition errors for program one becomes 0.75, and if the same outlier is removed for Program three, the average number of errors becomes 1.75, both of which are far more reasonable. As Table 5.2 illustrates, the average number of errors per person is
substantially higher than the average number of errors per native speaker, and substantially lower than the average number of errors per non-native speaker.

Regarding the UCP Study, the speech recognition errors decreased over time, as can be seen in Tables 5.3 and 5.4. The results for participant two improved dramatically as the initial session ended early because the participant’s meter of speech was not conducive for the speech recognition engine; however, after brief vocal training, participant two was able to change his meter and successfully build programs using Myna.

It is encouraging that user performance improved over time. Type A errors and Type D errors (both indicative of Myna performance) decreased over time in both studies. As previously mentioned, Type C (speech recognition) errors are a side effect and not the focus of this research; however, the Pilot Study data shows a decrease in the number of errors from Program One to Program Three, but the trend is inconsistent. Type B errors (incorrect Scratch command) occurred infrequently, but they are important to note. With more training, these errors should disappear. Type B errors did not occur during the UCP Study because the participants were given an exact script of the commands to say, and the participant who eventually stopped using a script had more training, which demonstrates that the target audience may require more initial training of programming concepts, which is not unexpected.

Overall, the Pilot Study participants felt Myna was “somewhat easy” to learn. The participants from both studies felt “satisfied” after using Myna and were not frustrated during the study. Additionally, Pilot Study participants felt that the vocal commands were “somewhat predictable” and that Myna performed the expected actions “most of the time.”

5.4.1. Threats to Validity

The primary threat to validity is the small participant group: five participants in the Pilot
Study and two participants in the UCP Study (total of seven) is a small sample size. Initially, three other UCP camp attendees showed interest in participating, but opted not to participate when prompted throughout the summer camp. Two of the three preferred to spend time with friends rather than work on the computer. This particular group of students does not have the opportunity to interact with each other very frequently; therefore, their main goal for the summer camp was to spend time with friends. The third attendee preferred to participate in a different activity by himself, and therefore, he chose not to participate in the study. Two additional attendees lacked the vocal articulation necessary to participate. This project requires training and observation over time, which means that it is a longer study and more difficult to recruit a large number of participants, especially for the target population. There are positives to this type of study (e.g., richer data, participant becomes comfortable with the observer and provides honest feedback) and negatives (e.g., lower number of participants).

5.4.2. Internal Validity

By asking each Pilot Study participant to create all of the programs in Scratch using the mouse/keyboard and then again in Myna using voice, there could be a threat to the validity of the time recorded when using Myna. Since the participant would know how to complete the program, this knowledge could reduce the time to recreate it using Myna. However, the aspect being timed was not how long the user took to successfully complete the logic of the program but, rather, how long it took the user to move the commands from the command palette to the command editor. Moreover, the user had to learn Scratch commands when using the mouse/keyboard and Myna commands when using Myna, which produced a learning factor in both scenarios.
Also, Pilot Study participants were given a two-minute introduction to Myna and no training in order to determine how easy Myna is to learn. However, this approach may have had a negative impact on the data as participants may have performed better and felt more satisfied with Myna after a longer training session.

5.5 Ability-Based Design Evaluation

To determine if Myna meets the design guidelines set forth by Wobbrock et al. (2011), we analyzed the survey data based on each category of Wobbrock’s definition. Only the first two categories are required, and the others are recommended. Therefore, it is not necessary for Myna to meet all seven guidelines, but it is a goal for which to aim.

1. Ability is defined as focusing on what users can do, which Myna does by utilizing speech rather than dexterous movement. To understand if users felt Myna met their abilities, we analyzed the Learning Myna category of the Pilot Study survey by looking at the median responses (see Figure 5.5 – Ability). The participants had a median response of 4/5 for each question, suggesting they felt “able” to use Myna. Additionally, the UCP Study participants both stated they would use Myna again on the satisfaction survey, and one participant stated, “[Myna] got somewhat easier the more I used it.”

2. Accountability states that the system should change, not the user. The solutions described in the Mapping section are an example of how we have adapted Myna to meet user needs. The median response (see Figure 5.5 – Accountability) varied between 3/5 and 5/5. 3/5 is lower than what we desire. The Parameters issue scored the 3/5, which matches with the number of errors observed (7% of errors, 1/14, were caused by Parameter issues). Although the functionality performed as expected, participants appeared to struggle with
how to use the vocal command properly. Simplifying the command from “set property one at” to “edit one at” will potentially improve this functionality.

3. Adaptation states that interfaces should be user-adaptable; thus, Myna requires a customizable command wizard, which would allow users to change vocal commands to better meet his/her vocal needs.

4. Transparency is defined as giving users awareness of what the application is doing. We reviewed questions regarding how users were provided information (see Figure 5.5 – Transparency) throughout the testing process. Only the category of error messages received below a four. The existing error messages were not invoked during testing (e.g., there were no errors starting the microphone); however, the users did experience errors, because of speech recognition issues, for which there were no messages. Further development is necessary to provide more error message feedback to the user. Also, one participant commented that it would be helpful if Myna displayed the command it interpreted from the user’s speech to better understand what Myna is doing.

5. Performance should monitor user actions and then predict behaviors necessary to meet those actions in the future. While Myna does not meet this goal entirely, Myna should have predictable commands allowing the user to feel he/she can predict the action associated with each command. Both questions related to this topic received a 4/5 (see Figure 5.5 – Performance); therefore, users found the commands to be predictable.

6. Context states that the system should sense the context and anticipate its effects. This ability is not yet built into Myna, but there are future plans to include it via a dynamic grammar (i.e., the grammar will change based on the user’s commands). For example, when the user chooses a particular menu, (e.g., “motion”) only the commands on the
selected menu will be available to the user (e.g., “move steps,” “go to x y”). This would also help prevent “ghost blocks” from being placed in the editor. A “ghost block” occurs when the user asks Myna to drag a block onto the screen that Myna misunderstands. The misrecognition will match a block from another menu, and Myna will attempt to move that block onto the editor. However, since that menu is not actually on the screen at the time, the block cannot physically be moved resulting in a “ghost block.”

7. Commodity is defined as being low-cost. Commodity is one of Wobbrock et al.’s (2011) most important requirements because individuals with motor impairments already have high costs associated with the adaptive technologies used to assist with their disability. Myna is a free tool and, other than a computer, requires only a microphone.

![Survey results per design guideline.](image_url)
Based on the analysis of each guideline, Myna meets three of the seven requirements set forth by Wobbrock et al. (2011): Ability, Performance, and Commodity. Future development will be focused on correcting the issues related to Accountability and Transparency in addition to adding the necessary features for Context (dynamic grammar) and Adaptation (customizable command wizard).

5.6 Lessons Learned from Evaluations

As Harris (2004) points out, anyone who needs a justification for usability testing of an application requiring human interaction might want to change careers. This is especially the case when the application consists of a GUI, or in the case of this research, a VUI. Although the test groups were small, the evaluations were enlightening and revealed the need for several additional features. Based on the feedback given and the work observed, the following lessons were learned about modifications needed for Myna (lessons were learned about the studies themselves, and these are communicated in Chapter 8, Future Work).

1. “set property at” is a confusing command and should be altered. For starters, this is a long phrase to say on a frequent basis. The word “edit” would be a more appropriate replacement.

2. The user should have the ability to customize the grammar. In the event that the user has difficulty pronouncing a particular word, or if the user simply wants to change a longer phrase (e.g., “drag and drop”) to a shorter one (e.g., “drop”), the user should have the flexibility to do so.

3. The user needs a method to “start over” in the event that the user either wants a new program or the speech recognition made a mistake. When the user selects “Edit > New” to begin a new program, the transparent frames and current program would still be in
memory in Myna. This is problematic because the user should not have to save, close out Myna, and then start again.

4. The user needs a method to “undo” in the event that he/she or the speech recognition made a mistake. “Undo” is not a native command within Scratch; only “undelete” exists. The user should be able to undo not only the Scratch work, but the vocal work as well.

5. The vocal command understood by the speech recognition engine should be displayed on the screen to increase user communication. If the speech recognition engine understands a different command than what the user said, the physical output will result in a different outcome than what the user expected, which is not good. If the command is displayed on the screen, the user could immediately comprehend why the outcome is different than expected.

6. Additional help messages that are context-oriented should be displayed on the screen. The current help messages are limited, and it could be useful to have more help messages displayed, particularly if those help messages related specifically to what the user was attempting to do within Myna.

These modifications are addressed in more detail in Chapter 6.
CHAPTER 6
ADAPTING MYNA BASED ON USER EVALUATIONS

Based on the evaluations described in Chapter 5, there are six changes that should be made to Myna in order to improve usability and better meet the design guidelines set forth by Wobbrock et al. (2011). The changes identified include parameter editing terminology (Section 6.1), grammar customization (Section 6.2), clearing the screen (Section 6.3), undo (Section 6.4), speech recognition display window (Section 6.5), and contextual help messages (Section 6.6). Wobbrock et al.’s (2011) intention was for applications to “sense” the user’s behavior. While Myna does not make adjustments automatically based on what the user does, the changes mentioned in this section act in the spirit of Wobbrock et al.’s (2011) design principles and provide user-adapted options.

6.1 Editing Parameters

In the second version of Myna, parameters were edited by stating, “set property ___ at __.” For example, if the user wanted to edit the second parameter in the fifth block (block labeled ‘5’) within the program (see Figure 6.1), the user would say, “set property two at five,” and Myna would direct the mouse to click on the second parameter slot within the third block in the program. The user would then have the ability to edit that parameter slot.

Based on the user evaluations, this command was too cumbersome. Sears et al. (2003) found that longer commands are less successful than shorter ones; thus, this command needed to be simplified. The “___ at _____” portion of the command is unavoidable, as the user must
specify both the parameter slot and the number of the block to be edited. Therefore, the only way to reduce the length of the command is to change “set property.” The command has now been changed so that the user need only say, “edit ___ at ____” (i.e., “edit two at five”).

According to Wobbrock et al. (2011), systems should be held accountable for change. If poor performance occurs, it is the fault of the system, not the user; thus, designers must respond by changing the system (Wobbrock et al., 2011). User evaluations indicated that the parameter editing functionality was not performing optimally for the user, and, following Wobbrock et al.’s guidelines, this was changed thereby increasing Myna’s accountability.

6.2 Grammar Customization

As one of the non-native English speakers evaluating Myna pointed out, it would be easier to use Myna if it were possible to edit some of the commands to better fit the user’s needs.
Thus, a dialog window was added to create a grammar customization wizard (Figure 6.2), which is invoked via command, “customize.” Unfortunately, this wizard requires the use of the mouse/keyboard due to the current limitation of not changing Myna’s grammar. There is no way to know what word the user might want to substitute for an existing command; therefore, without changing the grammar to a basic dictionary grammar (i.e., something that could be done in future work), the user must type the new command he/she wants to use as a replacement. The user will first select the command to replace from a dropdown list (Figure 6.2) and then type the new command. Figure 6.3 provides an example of how a user might change the “drag and drop” command to use “put” instead. It should be noted that, currently, only commands can be edited and not components due to the existing Myna design in how utterances are processed.

Figure 6.2. Dropdown list for customized grammar wizard.

Figure 6.3. Example use of customized grammar wizard.
Upon hitting ‘OK,’ Myna collects the command to replace and the new command from the dialog window. If the new command does not conflict with an existing command in the grammar file, Myna adds the pair to a Hashmap in the ScriptsState.java file. The new command is then mapped to the command to replace using Sphinx’s alias feature. Finally, in the main Myna.java file, there is an “if” statement that compares user commands to existing Myna commands. Within this “if” statement, the collection of new commands is traversed and compared to the user’s given command, and if a match is found, the appropriate method is invoked.

Regarding the adaptation design principle, it is important for systems to either be self-adaptive or user-adaptable (Wobbrock et al., 2011). Adding a grammar customization wizard allows users to adapt Myna to better meet their needs. This one feature may seem like a small gesture, but if the majority of what the user does is drag and drop blocks into the editor, allowing the user to change this command to something simpler could be extremely beneficial and save the user time, frustration, and vocal strain.

6.3 Delete Program and Start Over

The user should have the ability to start over in Myna and clear the screen. This was a fairly simple modification. A new command was added, “clear,” that will delete all of the blocks from the screen, clear out both the vector and Hashmap collections of components, and reset the initial drop point in the editor. If the component collections were not cleared, the transparent frames would remain on the screen; furthermore, if the drop point were not reset, the user’s new program would begin where the previous one ended. As a side note, if the user were to select “File > New,” the clear() method will be invoked to ensure the transparent frames (numbers) are removed from the screen and the drop point adjusted accordingly.
During the testing of this new feature, it was quickly made obvious that, for the sake of transparency, the user should be asked to verify that he/she wishes to clear the screen as it will result in the deletion of his/her current program. As a result, a dialog window (Figure 6.4) is now displayed when the user says, “clear,” to verify that the user intentionally made the command similar to the warning received when exiting a Microsoft Word document without saving one’s work. The default operation for the dialog window is “Cancel” as a precaution.

Wobbrock et al. (2011) define transparency as the ability to “inspect, override, discard, revert, store, retrieve, preview, and test adaptations.” By enabling users to essentially “preview” the results of a given command via the warning dialog, Myna is providing a layer of transparency to the user, which is a necessity for user-adaptive systems (Wobbrock et al., 2011).

6.4 Undo an Action

Speech recognition engines make errors; users do not (Harris, 2004). In fact, one out of every 20 spoken commands will be misunderstood by the speech recognition engine (Nass &
Brave, 2005). If one of these misrecognitions were to result in unwanted changes being made to the user’s program similar to unwanted changes described in (Sears et al., 2003), the user would become frustrated without an easy way to undo this change. Examples of unwanted changes might be the wrong block added or deleted, and while these would be simple changes for the user to recreate him/herself, the user should not have to make these changes because the system should.

“Undo” is a non-native command to Scratch; however, Scratch does have an “undelete” functionality. To implement “undo,” the Command design pattern (Gamma et al., 1995) was implemented. Furthermore, a stack of undoable Actions was first added to the AppState.java file. Actions have a type (e.g., “drag and drop,” “delete,” “drop after”), a component (i.e., the block being added/deleted), and a position (i.e., the position of the block as it appears in the program code). For any addition (e.g., “drag and drop,” “drop after,” “drop before”) or deletion, an Action is created and pushed onto the AppState stack. When the user issues the “undo” command, the top Action is popped off the stack and negated. If there are no Actions to undo, an appropriate dialog window is displayed to inform the user. Currently, there is no support for undoing parameter editing; this could be future work.

Giving the user the ability to “revert” actions via undoing them is an element of transparency (Wobbrock et al., 2011). By providing the user with the option to “undo” an action gives the user more power to control Myna with ease. For example, if a user were to accidentally delete the wrong block, the user would have to give the appropriate command (e.g., “drag…drop after,” “drag and drop…,” or “drag….drop before”), but having undo lets the user simply say, “undo.”
6.5 Speech Recognition Display

Graphical interfaces excel at providing feedback and often provide redundant feedback via sight, sound, and touch (Harris, 2004). Vocal interfaces, however, do not provide feedback well because voice is a transient input modality (Harris, 2004). Myna, unlike telephone voice interfaces, can take advantage of having a GUI as part of its application. It is the GUI element that allows Myna, a VUI, to provide better feedback to the user in the form of dialog windows.

In Chapter 5, it is demonstrated that there were 11 speech recognition errors per person across all three programs (this average does include one extreme outlier). These errors can cause frustration for the user if he/she is unaware that a speech recognition error has occurred; therefore, a display window showing the user what the speech recognition engine understood was added. It is a small dialogue window (Figure 6.5) that appears in the bottom right-hand side of the screen after each utterance (Figure 6.6). It closes after three seconds or after another utterance has been completed, meaning that it does not prevent the user from issuing his/her next command.

![Figure 6.5. Speech recognition display window.](image)
If an error occurs (i.e., the command understood by the user does not exist in the grammar), an error message will be displayed in the same dialog window (Figure 6.7) in order to keep the user better informed of what is happening behind the scenes. As mentioned in the previous section, it is to be expected for a misrecognition error to occur in one out of twenty utterances (Nass & Brave, 2005). It is important that the user does not feel at fault for this misrecognition, and similarly, it is important that the user not feel Myna is at fault to avoid the user losing confidence in him/herself or in Myna (Nass & Brave, 2005). Therefore, the message to the user is worded so as to blame neither the user nor Myna: “Command not understood.” A better approach would be to use scapegoating (i.e., blaming a fictitious third party such as ambient noise) (Nass & Brave, 2005), but this technique does not seem to be a good fit for Myna. To blame a scapegoat, there must exist a plausible third party to blame for the miscommunication between the user and Myna. Because the user is aware of all surrounding noise while using Myna, Myna cannot blame the misrecognition on ambient noise (i.e., the room could be perfectly silent). Another plausible scapegoat could be the wireless connection, but
Myna does not use a wireless connection. The microphone is part of the system; thus, it should not be the scapegoat. However, the microphone could be blamed for poor speech recognition if the user does not follow the recommendations and use a directional microphone, but Myna would not be aware of the type of microphone the user selects. In the example described by Nass and Brave (2005), a study for a voice-driven interface for a vehicle is conducted, and the interface blames any falter in the driver’s performance on the road (e.g., “Steering is difficult on this road;” “Turns on this road appear very quickly”). The Myna dyad consists of the user and Myna (includes the computer and microphone). There are no plausible third parties to use as a scapegoat.

While this is not an adaptation by Wobbrock et al.’s (2011) definition, transparency of Myna is improved by displaying the vocal command understood by the speech recognition engine. Users now have a method to “inspect” the vocal commands as they are given; thus, users have increased awareness of the data Myna is processing. This fails the spirit of ability-based design in that the speech recognition engine will not adapt to the user, meaning that if the user gives a command that the speech recognition misunderstands, then the user must adjust his/her pronunciation. As this research is not focused on the speech recognition aspect of Myna, improving the adaptability of Sphinx is beyond the scope of this research. The solution Myna offers is the grammar customization wizard discussed previously in Section 6.2.
A future improvement to the speech dialog window would be to add a customization component that would allow the user to edit the amount of time the window is displayed, which would increase the “adaptability” of Myna.

6.6 Help Messages

Although in the context of online help systems, Shneiderman and Plaisant (2010) state that “[t]he ability to provide context-sensitive information is a powerful advantage.” Context is an important aspect within a system in regards to accommodating a user. Sensing context at runtime is a difficult task, but it is a goal when designing a system (Wobbrock et al., 2011).

Myna handles context by providing specific informational or help messages based on the command given by the user. These help messages are displayed in the upper-right side of the screen for three seconds, but they do not interfere with any action performed by the user (i.e., the user does not have to wait for the dialog window to close before continuing with his/her program). Myna’s help messages were created with the philosophy that prompts should be short and simple (Kortum, 2008). Each help message is contained within 400 pixels; thus, the messages conveyed to the user must be short but complete. Furthermore, over prompting a user can be just as frustrating as under prompting a user; the level should be “[just right]” (Harris, 2004). Therefore, three instances within Myna were identified as potentially confusing for a novice Myna user.

1. When the program first begins, Myna alerts the user that voice recognition has begun.

Upon closing that dialog box, Figure 6.8 is displayed hinting to the user that he/she should select a menu and start dragging blocks onto the screen.
2. If a user wishes to place a block in between two existing blocks in the editor, the user must use the series, “drag….drop after…..” or “drag….drop before…..” Because this series involves two different commands, when the user invokes the “drag” command, an informational message will be displayed indicating the next step of the process (Figure 6.9).

3. It has already been stated that study participants found editing parameters to be difficult. When a user issues the “edit” command, a help message is displayed highlighting the next steps of stating a number or letter (Figure 6.10).

In the spirit of adaptation, Myna should provide the user with a way to lessen the amount of time help messages are displayed or to turn them off completely. Ultimately, Myna’s users would benefit from the addition of a system preferences file that would contain user preferences
such as the amount of time to display the speech recognition window and help messages. A preferences file would be beneficial to those users who become easily frustrated when presented with information they already understand (Cohen et al., 2004).

6.7 Conclusion

Although these changes have been made, additional user testing needs to be performed in order to evaluate the effectiveness of the changes. Additionally, these changes could be improved upon, particularly the grammar customization wizard and help messages. It would be optimal if the grammar customization wizard did not require the use of the mouse/keyboard. This would require switching to a dictionary grammar to capture the new word from the user and then returning to the Myna grammar with the new term added. Communication with the user is always important and can always be improved; thus, there must exist additional help messages that would be valuable to the user.

The above changes touched on four ability-based design principles: accountability, adaptation, transparency, and context (Wobbrock et al., 2011). By the system being changed upon the realization of poor performance, the “edit” command demonstrates Myna’s accountability. The grammar customization wizard adds user-adaptation to Myna. Transparency is the most prolific design principle evident in these changes as start over (“clear”), “undo,” and the display of what the speech recognition engine understands each touch on an aspect of transparency (e.g., “preview,” “revert,” and “inspect”). Finally, the context guideline is met via help messages being displayed based on user-issued commands.
CHAPTER 7
GENERALIZING MYNA

Creating a VUI for a GUI, which was not developed with the intention of mapping components on the screen, can be tedious and time consuming. In creating a VUI, both the static and dynamic components of the GUI must be captured. To streamline this process, a semi-automated approach was designed. Figure 7.1 illustrates the workflow of the three steps involved in this approach, and these three steps are outlined in the following sections.

Figure 7.1. The generalization of Myna for customized applications.
7.1 Importing the Static Structure of the GUI

A first step in speech-enabling a GUI is to understand all of the widgets that are of interest and must be clickable from vocal commands. There have been several approaches proposed to reverse engineer GUIs, based on techniques such as static analysis of source code (Staiger, 2007), dynamic execution of the application (Memon et al., 2003; Kumar & Sasikumar, 2008; Zettlemoyer & St. Amant, 1999), and reverse engineering of system resource files (ResHacker, 2004). The primary requirement of the screen scraping tool is to collect metadata about the different components on the screen, such as: the physical coordinates of the component, the dimensions of the component, location of any parameters, the user-defined name of the component, and the type of the component (e.g., is the component static or moveable?, is the component a container?). The approach utilized in this work is a semi-automated approach based on user input to conserve the effort needed to map the GUI fully.

Five tools providing some of the required functionality were reviewed in hopes of using a pre-existing tool to collect the application metadata: Tesseract (Smith, 2007), VisMap (Zettlemoyer & St. Amant, 1999), Sikuli (Yeh et al., 2009; Chang et al., 2010; Chang 2011), Prefab (Dixon & Fogarty, 2010), and PAX (Chang et al., 2011).

1. Tesseract (Smith, 2007): Tesseract, an open-source Optical Character Recognition (OCR) tool developed by HP between 1984 and 1995, was developed to improve the accuracy of HP scanners. In 1995, Tesseract proved to be a better OCR engine than commercial engines in the 1995 Annual Test of OCR Accuracy (Smith, 2007). Although it has not been updated in over a decade, Tesseract is a useful tool for capturing text; however, its accuracy could be improved significantly by using a Hidden-Markov-Model-based character n-gram model (Smith, 2007).
2. VisMap (Zettlemoyer & St. Amant, 1999): VisMap allows for programmatic control of an application through the GUI, and it builds a “structured representation of interface objects” by utilizing display input and image processing algorithms (Zettlemoyer & St. Amant, 1999).

3. Sikuli (Yeh et al., 2009; Chang et al., 2010; Chang, 2011): Sikuli Search allows users to search documentation databases using an image rather than text as the search item, and Sikuli Script allows the user to automate GUI elements on the screen by editing a Python script based on a screenshot of the application (Yet et al., 2009). Sikuli Script allows quality assurance testers to utilize images when writing scripts to test the application; the testers write the appropriate script, and Sikuli Script returns whether or not the automation performed according to the requirements (Chang et al., 2010).

4. Prefab (Dixon & Fogarty, 2010): Prefab reverse engineers the pixels of an application to evaluate the types of elements on the screen to allow for the addition of advanced functionality to any application. The primary work presented by Dixon and Fogarty (2010) compares element types found on the screen to that of a prototype library to determine the type of element found. Prefab searches the screen, creates hypotheses about the elements on the screen, and then tests the hypotheses. Dixon and Fogarty (2010) state that Prefab can reverse engineer non-standard interfaces that were not created using a toolkit in addition to those interfaces that were created using toolkits.

5. PAX (Chang et al., 2011): PAX is a framework that combines pixels with accessibility APIs to gather all metadata about an object. This framework improves the accuracy and the quality of the data collected about an object when combined with a pixel-based analysis tool (e.g., Sikuli) (Chang et al., 2011).
The metadata required for a semi-automated VUI creation tool are the physical coordinates of the component, the dimensions of the component, location of any parameters, the user-defined name of the component, and the type of the component (e.g., is the component static or moveable?, is the component a container?). This metadata should be collected without interfering with the underlying code of the application for which a VUI is being created. For example, when creating Myna, the Scratch code was never modified in order to create a flexible platform that could be applied to any GUI. This is the primary requirement for Myna; therefore, VisMap and PAX are not viable tools as both requires access to the application’s API. Another, similar tool was reviewed, Scrapy (2015), but it also depends on the API. According to Chang et al. (2011), OCR algorithms are better suited for scanning black text on white backgrounds, and Chang et al. (2011) states, “[u]sing current OCR algorithms on screen text would generate poor results.” Thus, Tesseract is not the best tool since it is an OCR engine.

Prefab and Sikuli Script are the best choices of these five (Sikuli Search provides a different functionality entirely as it is only used for searching). Both Prefab and Sikuli Script are based on the previous work of Zettlemoyer and St. Amant (1999) (Yeh et al., 2009; Chang et al., 2010; Chang, 2011; Dixon & Fogarty, 2010); however, the primary improvement is that neither Sikuli Script nor Prefab require access to the application’s API. Prefab differs from Sikuli Script because it finds all occurrences of all widgets on a screen, and it does so significantly faster than Sikuli (Dixon & Fogarty, 2010); however, Prefab only finds common widgets. The applications for which VUIs are being proposed in this research do not use typical widgets (e.g., buttons, checkboxes, textboxes) used by most application development toolkits (e.g., Visual Studio, NetBeans) (Dixon & Fogarty, 2010). Sikuli Script identifies a component on the screen via a screenshot of that component (provided by the user), and then, that component can be automated
through the user-defined script (Yeh et al., 2009; Chang, 2011). Based on our previously defined requirements, neither application returns the information required; however, Sikuli Script might provide some benefits for modeling the behavior of a GUI.

Although each of these tools provides functionality similar to what is needed for Myna generalization, none of them fully meet the defined requirements; therefore, after further research, a custom tool has been developed: MynaMapper, which is described in the next section.

7.1.1. MynaMapper Description

MynaMapper, written in Java, allows a user to map graphical components on the screen and store the components’ information in property files, which are later integrated into the Myna framework. Depending on the number of graphical components on the screen, this can result in a large number of files, or a small number of files. For example, Myna was originally developed for Scratch v1.4, and contains 168 property files. When manually creating a VUI for this type of application, the user must create each individual property file, which can contain anywhere from five to 13 values. The majority of these values consist of xy-coordinates, which can be time consuming to collect. Instead of taking screenshots or having to hover the mouse over a particular location to garner the necessary information for each required value, MynaMapper allows the user to simply click through the application (in a specified order) and store the automatically collected information.

As an example, in Scratch, if a user were to collect information regarding a movable component on the screen (i.e., a component which can be dragged and dropped), six to nine values should be collected: the name of the component, the parent of the component, the x-value of the initial location, the y-value of the initial location, the length of the component, the height...
of the component, and the location of each parameter (there are zero to three parameters for any movable component in Scratch). The time-consuming portion of this process is collecting the x- and y-values. MynaMapper simplifies this by providing a simple GUI for the user to complete for each movable component (see Figure 7.4).

Upon starting MynaMapper, the user is presented with a textbox to indicate the directory in which to store the subsequently created property files (Figure 7.2). The user is also asked to identify the types of components that will be mapped and the properties to be collected for movable components (e.g., length, height, parameter1, parameter2, etc.). Upon submitting this information, the user is then directed back to the main application page and will click on ‘Collect Information’ in the menubar to begin collecting the desired information (Figure 7.3). This menu contains all of the component types the user selected in the previous screen (Figure 7.2). In the pictured example, only ‘Command Palette’ and ‘Movable Component’ were selected.

![Figure 7.2. Initial MynaMapper screen when starting a New program.](image-url)
Figure 7.3. Collect Information menu.

When the user selects an option from the ‘Collect Information’ menu, a corresponding dialog window appears that allows the user to capture the specified information. Figure 7.4 is an example of the ‘Movable Component’ window. To complete this simple GUI, the user only needs to type in the name of the block and the name of the parent of the block (i.e., any menu that must first be selected prior to reaching the movable block). The user then clicks on ‘Get Grab Point’ (this is a mandatory field), which then projects a transparent frame covering the whole screen. When the user clicks on the location where the user would normally try to click on the component to drag and drop it into the editor, MynaMapper captures the xy-coordinate of where the user clicked. MynaMapper then stores the xy-coordinates of this “grab point” as the x-value of the block’s initial location and the y-value of the block’s initial location. Figure 7.5 shows the MynaMapper interface being used to get the “grab point” of ‘move steps.’ The user would need to click anywhere on the ‘move steps’ block.

To get the length and height of the component, the user clicks on ‘Get Length and Height,’ a transparent frame is again projected across the screen, and the user proceeds to draw a rectangle around the desired component on the screen. Parameter locations are stored as

Figure 7.4. MynaMapper data collection screen for a movable component.
distances from the original grab point. To collect this information, the user enters the number of parameters for the component, clicks on ‘Get Parameters,’ and then clicks on the parameter locations within the component. After the user clicks submit, all of the information is calculated appropriately, and a property file is written and stored in the previously specified location. This process is then repeated for all types of components necessary to enable a VUI for a desired application, and the resulting property files are integrated into the pre-existing Myna platform.

7.1.2. Initial MynaMapper Testing

In order to evaluate the amount of time saved by using MynaMapper rather than manually creating the necessary property files, the property files for Scratch v2.0 and Lego Mindstorms were first created manually while documenting the time taken to create them. Next, MynaMapper was used to create the property files and the time documented. The times were then compared and are presented in Table 7.1.
Table 7.1. A comparison of the times taken to create the necessary property files.

<table>
<thead>
<tr>
<th>Application</th>
<th>Manual Creation</th>
<th>MynaMapper</th>
<th>Decrease in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch v2.0</td>
<td>600 minutes</td>
<td>41 minutes</td>
<td>93%</td>
</tr>
<tr>
<td>Lego Mindstorms</td>
<td>960 minutes</td>
<td>15 minutes</td>
<td>98%</td>
</tr>
</tbody>
</table>

The first test involved manually creating property files for Scratch v2.0\(^8\), an online version of Scratch v1.4, which is what Myna was originally designed to control. The manual process involved taking screenshots and then using MS Paint to determine the $xy$-coordinates. It took approximately 10 hours to map all of the components on the screen and create the corresponding files. In the initial testing of MynaMapper, the semi-automated approach took less than 41 minutes to create the 118 required property files. This evaluation reflects a 93% decrease in the time taken to create the required property files. During the initial test run, four user errors occurred, which have been addressed by adding a menu item that allows the user to see a list of all property files created by category (e.g., movable block, movable block container, or static component) as well as the ability to delete or recreate these files.

As a second test, the property files required for Lego Mindstorms were manually created, which took roughly 16 hours to map and test the files (960 minutes). The initial manual mapping only took six hours, but a different technique was used (a short Java program was used to read the $xy$-coordinate as the mouse hovered over the desired components). This technique resulted in several errors, and the files had to be recreated, which took an additional ten hours. It took MynaMapper 15 minutes to map the same files resulting in approximately a 98% decrease in the amount of time taken to create the required property files. Mindstorms is a little trickier than Scratch v2.0 because Mindstorms has a different configuration for the “parameters” than Scratch.

---

\(^8\) Videos of Scratch v2.0 being programmed by voice can be found at https://www.youtube.com/user/ProjectMyna.
To accommodate this difference, MynaMapper allows the user to specify the necessary properties to collect when creating a “New” project. Only two user errors occurred during the second test showing a slight improvement from the first experiment in the Scratch migration.

7.2 Specifying the Dynamic Execution Behavior of the IPE

Aside from the static properties of the IPE screens, the dynamic behavior must also be captured. This represents the valid execution states of the interaction among various screens or modes of an application. For example, the user of a speech-enabled application should only be allowed to vocalize commands that make sense in the current context. The VUI developer must specify all of the interactions among the execution paths of the IPE. There are two possible approaches:

1. The use of a modeling language to capture such interactions. Domain-specific software environments (Gray et al. 2007) are emerging as a powerful tool for rapid development of software from high-level visual models. Metamodeling tools have been shown to maximize reuse of domain knowledge by capturing an appropriate level of task abstraction that can be easily and efficiently used to synthesize new applications (Gray et al. 2007).

2. A simple Domain-Specific Language (DSL) could be created to model the behavior that differs from the base platform (designed for Scratch v1.4) to the new application in the browser.

Option two provides all of the necessary functionality needed to capture the dynamic behavior changes from one application to another; therefore, a simple DSL, MynaScript was designed. MynaScript was written using the Xtext Eclipse plug-in (Xtext, 2015) and is presented in Listing 7.1.
Listing 7.1. MynaScript DSL.

```plaintext
grammar org.xtext.example.myscript.MynaScript with org.eclipse.xtext.common.Terminals

generate mynaScript "http://www.xtext.org/example/myscript/MynaScript"

MynaScript:
  sequence += Action*

Action:
  'action' name = ID
  '{'
  (movements += Movement)+
  '}'

Movement:
  name = Event + '.' + (value = Value)?

Event:
  'press' | 'release' | 'doubleClick' | 'pressDelete' | 'moveTo' | 'moveDropPointUp' | 'moveDropPointDown'

Value:
  'componentXY' | 'initialXY' | 'dropPointXY' | 'componentHeight' | 'componentWidth' | 'trashXY'
```

The DSL is simple, yet it provides all of the necessary components for describing behaviors (e.g., “Delete” and “Drag and Drop”) within an IPE. The user will define a method using ‘action,’ which will be composed of ‘movements.’ Each ‘movement’ contains a ‘name’ corresponding to an ‘event’ with an optional ‘value’ (see Table 7.2 for a description of what each ‘value’ represents). The purpose of the DSL is to capture differing behaviors among IPEs. A review was conducted of various IPEs: Scratch v1.4, Scratch v2.0, Lego Mindstorms, App Inventor, and Code.org, revealing very few differences in their programming behavior. The primary differences were how blocks were connected (vertically versus horizontally) and how
blocks were deleted, which is why these two activities were selected as the testing for MynaScript.

As an example, if the user wanted to write the “delete” or “drag and drop” methods using MynaScript, the user would only need to write the corresponding five lines in Listing 7.2.

Listing 7.2. MynaScript of “delete” and “dragAndDrop” methods for Scratch v1.4 or Scratch v2.0.

```plaintext
action delete{
    moveTo.componentXY
    press.
    moveTo.trashXY
    release.
    moveDropPointUp.componentHeight
}
action dragAndDrop{
    moveTo.initialXY
    press.
    moveTo.dropPointXY
    release.
    moveDropPointDown.componentHeight
}
```
While some of these commands are similar to those in Myna (i.e., “press” is equivalent to “robotMousePress()”), these commands do not require the user to know very much about Myna, the Java Robot, or the pre-existing methods used in Myna. Moreover, the generality of these methods lend very easily to converting to different IPEs. The scripts in Listing 7.2 are for Scratch v1.4 or Scratch v2.0, they would be interchangeable; however, they would be slightly different for Lego Mindstorms (see Listing 7.3). The only differences are that the “delete” key is pressed rather than dragging and dropping the components in a “trash” location, and the components are connected horizontally rather than vertically. Although the user would need to know these behavioral differences, the user does not need to know the specifics of how those behaviors are modeled in the code.

Listing 7.3. MynaScript of “delete” and “dragAndDrop” methods for Lego Mindstorms.

```java
action delete{
    moveTo.componentXY
    pressDelete.
    moveDropPointUp.componentWidth
}
action dragAndDrop{
    moveTo.initialXY
    doubleClick.
    moveTo.dropPointXY
    release.
    moveDropPointDown.componentWidth
}
```

Table 7.3. Lines of code (loc) needed to write “delete” and “drag and drop” methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>IPE</th>
<th>MynaScript (loc)</th>
<th>Myna (Java) (loc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>“delete”</td>
<td>Scratch v1.4/v2.0</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>“drag and drop”</td>
<td>Scratch v1.4/v2.0</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>“delete”</td>
<td>Lego Mindstorms</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>“drag and drop”</td>
<td>Lego Mindstorms</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>
To demonstrate the amount of time saved by using MynaScript, Table 7.3 lists the number of lines of code needed to write the “delete” and “drag and drop” methods in both MynaScript and Java. On average, there is a 57% decrease in the number of lines of code the user must write to describe these behaviors.

### 7.3 Code Generation of Grammars and Programmatic Control Code

After the static structure and dynamic behavior of the IPE have been obtained, the grammar needed by the speech recognition system can be generated as well as the programmatic control of the mouse and keyboard through the Java Robot Class.

**Listing 7.4.** Generated code from MynaScript.

```java
public void delete(AbstScratchComponent component) {
    ScriptsState scriptsState = AppState.getCurrentScriptsState();
    robot.mouseMove(component.getX(), component.getY());
    robotMousePress();
    robot.mouseMove(trash.getX(), trash.getY());
    robotMouseRelease();
    int pos = scriptsState.getMovableComponents().indexOf(component);
    removeComponentAt(component, pos);
    int adjustment = component.getHeight() * -1;
    moveCurrentDropPoint(adjustment);
}

public void dragAndDrop(AbstScratchComponent component) {
    ScriptsState scriptsState = AppState.getCurrentScriptsState();
    robot.mouseMove(component.getX(), component.getY());
    robotMousePress();
    robot.mouseMove(scriptsState.getDropX(), scriptsState.getDropY());
    robotMouseRelease();
    int currentX = scriptsState.getDropX();
    int currentY = scriptsState.getDropY();
    moveCurrentDropPoint(component.getHeight());
    scriptsState.getMovableComponents().add(component);
    component.setX(currentX);
    component.setY(currentY);
}
```
MynaMapper generates the grammar file; thus, all that remains for this step is to generate the code for the behaviors identified in Section 7.2. A code generator, using Xtend (2015), was added to MynaScript so that as the user creates the actions using the DSL, a .java file is created with the corresponding code for Myna. Listing 7.4 is the generated code based on the scripts written in MynaScript from Listing 7.3. Once generated, these methods can replace the existing methods within the “CommandExecutor.java” file in Myna.

7.4 Conclusion

There are three elements Myna needs when transitioning to a new IPE: the new grammar file, the new property files, and code for differing behaviors. MynaMapper produces all of the property files as the user maps them, and MynaMapper generates the grammar file after the user has finished mapping all of the required components. After the user has finished creating all of the mappings, these files can then be placed in the resource folder of Myna. Next, the user writes the methods for any differing behaviors using MynaScript. As previously mentioned, these behaviors will most likely be “delete” and “drag and drop” (if blocks are connected horizontally rather than vertically, “drop after,” “drop before,” and “drop in” will also need to be written, but these methods were not included in our testing of MynaScript). Next, the user will replace the methods in “CommandExecutor.java” with the corresponding methods auto-generated based on what the user wrote in MynaScript.

Although the approach presented here (using MynaMapper and MynaScript) requires human interaction, it does reduce the amount of time it takes to perform the same tasks manually. By using MynaMapper, the user can save up to 98% of the time it would take to manually map all of the files and create the grammar. MynaScript can then reduce the amount of code the user may have to write by 57%.
CHAPTER 8

FUTURE WORK

In addition to making further enhancements to Myna to better meet ability-based design guidelines and adding more behavioral components to MynaScript, there are many directions in which Myna can grow. In particular, more evaluations could be performed on the VUIs created using the Myna Framework beyond usability testing. Also, the Myna Framework can be applied to additional applications, different input modalities could be added to the framework, and more user studies would be beneficial to the overall project. These avenues of future work are described in the following sections.

8.1 Evaluating Newly Mapped VUIs

VUIs are similar to GUIs, but the only modality for interaction is a user’s voice. A system may respond back to a user either vocally (audibly) or visually. The number of existing techniques for testing VUIs (Brome, 2015; Cohen et al., 2004; Dybkjær & Bernsen, 2001; Farinazzo et al., 2010; Goss & Gilbert, 2007; Harris, 2004; Kaiser, 2006; Klemmer et al., 2000; Kortum, 2008) is an order of magnitude less than what exists for GUIs. Many techniques for improving the testing of GUIs involve efficiently creating more complete test suites (Amalfitano et al., 2012; Bryce & Memon, 2007; Memon, 2008; Memon et al., 2005; Nguyen et al., 2010; Xie & Memon, 2008; Yin et al., 2005). It is possible that some of these same techniques can be used to test VUIs in order to allow for faster and more thorough evaluations. Some existing GUI testing techniques are discussed in Section 8.1.1, existing VUI testing techniques are presented in
Section 8.1.2, Section 8.1.3 compares and discusses possible overlap of GUI and VUI testing, and Section 8.1.4 provides future work for evaluating both a GUI and a VUI in relation to Myna.

8.1.1. Existing GUI Testing Techniques

The critical component in testing software is to develop a comprehensive test suite. GUIs have an exponential amount of possible interactions requiring a very large number of test cases in order to conduct comprehensive testing on how well the GUI works; however, a common type of tool used in GUI testing, capture/replay tools, results in creating only a small number of test cases causing insufficient testing (Xie & Memon, 2008). In addition to the challenge of creating test cases, the test cases must be maintained and executed; this is where automation plays an important role because maintaining and executing the test suite is expensive (Memon, 2008; Memon et al., 2005; Nguyen et al., 2010; Xie & Memon, 2008). Although there are a variety of techniques for maintaining and executing test suites, selected techniques are discussed below in two categories: regression testing and fault detection. A recurring theme throughout the research is that test cases should be generated automatically (minimizing human interaction) (Amalfitano et al., 2012; Memon et al., 2005; Xie & Memon, 2008).

Regression Testing

Having an accurate test suite is a key component of using regression testing, but the test suite can be difficult to maintain (Memon, 2008; Yin et al., 2005). Maintenance can be complicated because GUIs are commonly developed using rapid prototyping (Memon, 2008), which involves many iterations of a GUI resulting in unusable tests within the test suite. Updating the test suite can become expensive; thus, using an automated approach for repairing the test suite, like that presented by Memon (2008), could be beneficial. Memon (2008) presents an automated approach for repairing the unusable tests, allowing a tester to then rerun these tests on the newly modified GUI. Similarly, Yin et al. (2005) present an Actionable Knowledge
Model utilizing an agent-based framework to maintain and repair test suites. Another issue with the regression testing of GUIs revolves around software being updated nightly. Often, the GUI testing of nightly updates is forgotten (Memon et al., 2005). DART (Daily Automated Regression Tester), an automated regression testing framework, addresses this issue (Memon et al., 2005). DART allows GUIs to be smoke tested (i.e., initial testing after modification to ensure an application works) automatically on a frequent basis and at a low cost (Memon et al., 2005).

**Fault Detection**

Similar to regression testing, the key component in fault detection in GUIs is an adequate test suite. Fine tuning the test suite through prioritization (Bryce & Memon, 2007) or refining existing models (Xie & Memon, 2008) allows for improved GUI testing. Bryce and Memon (2007) evaluated test suite prioritization using different methods (e.g., unique event coverage, Event Interaction Coverage (EIC), longest to shortest, shortest to longest, and random test ordering) finding that using EIC to prioritize the test suite “increases the rate at which faults are detected during GUI testing.” In a later work, Xie and Memon (2008) refine the event-flow graph (EFG) creating an event-interaction graph (EIG), which is reduced using minimal effective event context (MEEC) thereby creating fewer test cases. Using the resulting test suite, Xie and Memon (2008) evaluated the EIG test suite using four open source software applications, which found bugs not previously reported yet still relevant; thus, the EIG created test suite provided a more comprehensive test suite than those previously utilized.

**8.1.2. Existing VUI Testing Techniques**

There are a variety of testing techniques applied in industry including Dialog Traversal Testing (DTT) (Cohen et al., 2004; Kaiser, 2006), system quality assurance testing (Cohen et al., 2004), load testing (Cohen et al., 2004; Farinazzo et al., 2010), recognition testing (Cohen et al.,
2004; Farinazzo et al., 2010; Harris, 2004), evaluative usability testing (Brome, 2004; Cohen, et al., 2004; Dybkjær & Bernsen, 2001; Kortum, 2008), Wizard of Oz (WOZ) testing (Brome, 2015; Dybkjær & Bernsen, 2001; Goss & Gilbert, 2007; Harris, 2004; Klemmer et al., 2000; Kortum, 2008), usability inspections (Farinazzo et al., 2010; Harris, 2004), VUI Review Testing (VRT) (Kaiser, 2006), surveys (Goss & Gilbert, 2007), call recordings (Brome, 2015; Cohen et al., 2004; Goss & Gilbert, 2007), and call logs (Brome, 2015; Cohen et al., 2004). These evaluation techniques are used at different phases during the development life cycle. However, system designers apply the majority during the design or evaluation phase of the development life cycle; only surveys, call recordings, and call logs are exercised after deployment. Four techniques (e.g., Wizard of Oz testing, VUI review testing, usability inspection, and prototype testing) are discussed in more detail below.

**Wizard of Oz Testing**

“The Wizard of Oz is not only a classic movie, but also a testing methodology through which you can tell your application is on the right road to Kansas” (Brome, 2015). Wizard of Oz (WOZ) testing is a very common testing technique used primarily during the design phase of the development life cycle (Dybkjær & Bernsen, 2001; Goss & Gilbert, 2007; Harris, 2004; Kortum, 2008), but it can also be useful to a developer during the test and analysis phases (Klemmer et al., 2000). It consists of a “wizard” who reads system prompts based on the call flow design (Goss & Gilbert, 2007). Similar to prototype testing (Harris, 2004; Kortum, 2008), WOZ testing allows designers to evaluate the program flow (i.e., test call flow, grammar, prompts) without investing a significant amount of resources into the project (Goss & Gilbert, 2007).
Prototype Testing

While similar to WOZ testing, prototyping takes testing to the next level as it involves using an actual system prototype (Kortum, 2008), which implies that coding has been performed. It is important to test using a functioning prototype and has been referred to as the “gold standard” in evaluating an application’s usability and results in the amount of data collected being “an order of magnitude more realistic than the data from a WOZ test” (Kortum, 2008). Kortum (2008) expresses the opinion that WOZ testing is more resource intensive than at first glance and that the “benefit is exaggerated” relative to the amount of work it requires when compared to the cost-benefit of prototype testing.

VUI Review Testing

VUI Review Testing (VRT) is a type of “holistic” and “experiential” review where a VUI designer reverses roles and acts as the caller of the system exercising each use-case scenario previously determined (Kaiser, 2006). It is important that the tester be a VUI designer or usability specialist and not a developer because the goal of VRT is to verify the quality of the user experience (Kaiser, 2006). VRT should be conducted after DTT has already been performed but before User Acceptance Testing (UAT) in order to catch possible usability problems, prompt quality problems, and to determine if and where pauses might need to be placed or extended in addition to other, more general, problems (Kaiser, 2006). Because VRT catches these types of issues, it results in more informative usability testing.

Usability Inspection

Usability inspections are more empirical in nature and, thus, more objective (Dybkjær & Bernsen, 2001; Farinazzo et al., 2010; Harris, 2004); however, the number of established metrics for evaluating VUIs is minimal (Kortum, 2008). Dybkjær and Bernsen (2001) created a template
consisting of 15 usability issues on which the usability of a VUI should be evaluated, but many of these criteria are evaluated subjectively rather than objectively. Farinazzo et al. (2010) also presented a checklist type methodology for a heuristic evaluation. Both Dybkjær and Bernsen (2001) and Farinazzo et al. (2010) feel a usability expert should be the one performing the evaluations. Evaluation criteria resulting in more empirical/objective data might consist of the number of interaction problems, proper entry recognition rates, mean recognition scores, task success ratio, timeouts, number of recognition by type (e.g., rejection, deletion, insertion, substitution), and early terminations (Dybkjær & Bernsen, 2001; Farinazzo et al., 2010; Harris, 2004).

8.1.3. Combining GUI Testing Techniques with VUI Testing Techniques

While some VUI testing techniques are similar to those in GUI testing, VUI testing presents new challenges due to the temporary nature of the voice input modality. Many of the existing techniques for VUI testing focus on Interactive Voice Response (IVR) systems, which are the telephone systems companies use to manage customer calls. These vary greatly from a GUI where the input modality is voice rather than the mouse/keyboard. While the input modality is temporary, the feedback is permanent. This type of VUI is a mutant of a GUI and the VUI referenced in the aforementioned VUI testing techniques. The primary difference between GUI and VUI testing is that GUI testing appears to be more empirical (objective) (Amalfitano et al., 2012; Bryce & Memon, 2007; Memon, 2008; Memon et al., 2005; Nguyen et al., 2010; Xie & Memon, 2008; Yin et al., 2005) where VUI testing appears to be more user-focused (subjective) (Brome, 2015; Cohen et al., 2004; Dybkjær & Bernsen, 2001; Farinazzo et al., 2010; Goss & Gilbert, 2007; Harris, 2004; Kaiser, 2006; Klemmer et al., 2000; Kortum, 2008). VUI testing would benefit from more objective testing (Dybkjær & Bernsen, 2001; Farinazzo et al., 2010;
A method to add more objectivity to VUI testing would be to take advantage of existing GUI techniques. Based on the research previously discussed, this would mean developing more comprehensive test suites, automatically, which would evaluate the success of the application based on set requirements rather than subjectively.

Regardless of the technique borrowed from GUI testing, WOZ testing and prototype testing would still be necessary to capture the vocal portion of the application; although, WOZ testing in this mutant VUI is a minimized version. Prototype testing would be more appropriate than WOZ testing other than to evaluate phrases during the initial design phase. VRT and usability inspections would be helpful to ensure a fully comprehensive evaluation were performed, particularly from the user’s perspective. Any other testing performed would add to existing testing in order to examine the behavior of the mutant VUI.

One possible strategy for creating a test suite might be to use the commands in the grammar as if they were buttons and textboxes. The grammar would have to be written in such a way that commands (e.g., file, edit, new, close) would be separated from terminals (e.g., number, 0, letter, a). Moreover, the grammar should be written so that a parser could depict the type of each verbal command (i.e., distinguish a menu from a menu item). For example, the following could be a subset of a grammar where the word on the left side of each row (following an action command) represents the menu, and any words thereafter represent a menu item.

```java
public <click_action> = <click> (file | new | open | save | quit |
   edit | copy | cut | paste);
```
Figure 8.1. EIG for Open menu item based on grammar example (Xie & Memon, 2008).

An EIG (Xie & Memon, 2008) could then be created and a comprehensive test suite developed. Figure 8.1 represents the EIG for the Open menu item. Once the test suite is created, it could then be prioritized using event interaction (since an EIG would be used to create the test suite, there may not be more prioritization necessary) (Bryce & Memon, 2007) possibly reducing the number of test cases. The ability to repair test suites for regression testing would be important, and the technique introduced by either Memon (2008) or Memon et al. (2005) could be used. If nightly updates were necessary for the VUI, DART (Memon et al., 2005) would be the most appropriate tool. Figure 8.2 depicts a possible development lifecycle for a VUI with testing components added.

8.1.4. Applying VUI Testing to Myna

Although the Myna framework has moved well beyond the design phase, the resulting VUIs have not. There are numerous IPEs for which Myna could be used to develop a VUI, and some of the techniques discussed in Section 8.1.2 could be used during the design and testing phases of the development life cycle. Because WOZ testing is primarily for IVR systems, the effort it takes to set up a WOZ test does not seem worth the benefit; however, something similar where the
grammar could be evaluated during the design phase would be important. VRT and usability inspections would be very applicable and should be performed. VRT, the testing of use-case scenarios, would be easy to apply to a VUI created using Myna. Empirical data from usability inspections (e.g., recognition rates, success ratios, timeouts) would provide an objective evaluation of the resulting VUI. Moreover, creating test cases using an EIG (Xie & Memon, 2008) would provide an additional objective evaluation of the VUI. A system for running these types of evaluations should be added to the Myna Framework for future developers to utilize.

8.2 Future Extensions of Myna

Chapter 7, Section 7.1 describes extending Myna to other IPEs, specifically work that has been done regarding Scratch v2.0 and Lego Mindstorms. Throughout the 2014-2015 academic year, progress was made in configuring Myna for Code.org’s “Angry Birds” game (Code.org,
2015), which is described in Section 8.2.1, and the future work of applying Myna to Blockly created IPEs (e.g., Pixly and Spherly) is discussed in Section 8.2.2.

8.2.1. Code.org

The Code.org site contains several different Blockly constructed IPEs to learn programming basics (Code.org, 2015). Throughout the 2015-2016 academic year, two undergrads from the University of Alabama’s Computer Science department made progress manually mapping Myna to the “Angry Birds” program on the Code.org site (Timberlake, 2015). Each IPE on the Code.org site consists of programming puzzles of various levels of difficulty, and each level focuses on a specific lesson (Code.org, 2015). As the levels/lessons increase, there are new blocks added to the screen. This was only partially addressed in the work performed by the two undergrads. Moving forward, it would be more advantageous if each of these levels were incorporated into the Myna VUI.

As previously mentioned, there are numerous programs on the Code.org site (Code.org, 2015), but only the “Angry Birds” program was mapped to Myna. It would be interesting to use

![Figure 8.3. “Angry Birds” program from Code.org exemplifying a grey block, which cannot be deleted](Code.org, 2015).
the Myna Framework to map all Code.org puzzles to Myna. This might require additional behaviors added to the MynaScript DSL. For example, in “Angry Birds,” some levels already have blocks in the text editor (see Figure 8.3). It would be convenient if MynaScript had a built-in behavior to accommodate for these types of puzzles.

8.2.2. Blockly: Pixly and Spherly

As mentioned in Chapter 1, an undergraduate researcher at the University of Alabama utilized Blockly (2015) to create Pixly and Spherly (Trower & Gray, 2015). Both of these IPEs possess simple, block-based GUIs, which would be perfect specimens for which to create VUIs. Additionally, to better evaluate MynaMapper and MynaScript, it would be best if these IPEs were mapped manually and then mapped again using MynaMapper and MynaScript. This would provide an additional baseline for evaluating the efficiency of MynaMapper and MynaScript.

8.3 Future Input Modalities

In future revisions of Myna, other input modalities (e.g., eye tracking, eye gaze, face pose, touch) should be analyzed as possible input modalities to add to Myna in combination with voice making input multimodal. It is important to note that this should only be done while keeping costs as low as possible in order to maintain Wobbrock et al.’s (2011) seventh design principle of Commodity. Multimodal programming environments exist (Wilson, 2004; Ryokai et al., 2009) as do more general multimodal applications (Zapata, 2014; Slaney et al., 2014). Wilson (2004) proposed a multimodal programming environment for Dyslexic students that allows a user to add programming components via voice in addition to hearing code dictated from the computer in hopes that a multimodal approach will help Dyslexic students better comprehend the programming process. Ryokai, Lee, and Breitbart (2009) also present work on a multimodal programming environment; however, their focus is on an environment for young
children: the Pleo “Thought Bubble.” This environment combines touch with a GUI to allow children to program the UGOBE Pleo robot (Ryokei et al., 2009). In more recent research, Zapata (2014) and Slaney et al. (2014) demonstrated success in combining other input modalities with voice. Speech plus touch (Zapata, 2014) provided for improved Translator-Computer Interaction (TCI). Eye gaze and face pose could provide a better second input modality and has been shown to be successful (Slaney et al., 2014). Based on our target audience, touch is not the most viable input modality for our work with motor impairments; however, if it is an additional input modality option alongside voice and possibly eye tracking, it could be beneficial to those users within the target population who may have more dexterity than others.

8.4 Future User Studies

The user studies described in this dissertation involve a very small number of participants. In order to evaluate the usability of a VUI created using the Myna Framework more thoroughly, additional user studies should be conducted. Lessons learned from the previous user studies are listed below.

1. The study should be expected to take a minimum of six weeks. The target user study discussed in Chapter 5 involved meeting twice per week for five weeks. It was observed that an additional week would have been beneficial to allow the participants more time using Myna to create a custom program.

2. Meeting multiple times per week with each session lasting at least 30 minutes is ideal. In order for the participant to retain information from one session to the next, meeting multiple times each week is necessary. Also, the sessions should last long enough for the participant to complete the desired skill, but the session should not be so long that the participant loses interest.
3. The duration of the session should be dependent on the participant’s needs (e.g., attention span, interest, fatigue level).

4. The first session (possibly the first few sessions) should focus on dictation and reviewing the participant’s ability to articulate the VUI’s grammar.

5. The participant should have a desire to learn how to program (i.e., it would be more interesting for the participant).

6. Log the sessions electronically to capture the vocal commands the VUI understood.

7. Video tape the sessions (unless this causes undue stress to the participant) to collect the vocal command the user spoke, which could then be compared to what the VUI understood.

By following these suggestions, future user studies may provide more valuable feedback and would be more beneficial to the participant. Possible future participants could consist of:

1. classroom using Code.org during the “Hour of Code” (this would not be a longitudinal study, but it could provide some value);

2. disabled veterans; and

3. inclusive classroom environments involving grades 6-12.

8.5 Conclusion

The research presented in this dissertation has inspired four possible areas of future work. First, techniques similar to those used to test GUIs could be developed to VUIs created using Myna. There are several testing techniques currently in use for testing VUIs such as IVR systems, but these have not yet been applied to VUIs like Myna. Second, Myna can be used to create additional VUIs for other applications such as Code.org’s Hour of Code, Pixly, and Spherly. Third, Myna is currently dependent on voice but could be adapted to allow multimodal
input. Fourth, additional user studies would provide more information regarding the usability of Myna.
CHAPTER 9
CONCLUSION

It would be beneficial for students of all ages to be made aware of the opportunities available in Computer Science. The best approaches that have been proposed thus far are based on programming environments designed specifically for initial learners that are inquiry-based (Carlisle et al., 2005; Gray et al., 2014; Kelleher & Pausch, 2005; Malan & Leitner, 2007; Papert, 1980; Resnick et al., 2009; Wolber, 2011), motivate the learning experience from a specific context (Barbosa et al., 2008; Fenwick et al., 2011; Goadrich & Rogers, 2011; Kelleher & Pausch, 2005; Mahmoud, 2011; Uludag et al., 2011; Wolber, 2011), and assist in reducing the burden of learning the details of concrete syntax (Kelleher & Pausch, 2005; Malan & Leitner, 2007; Papert, 1980; Resnick et al., 2009; Wolber, 2011; Roy, 2012). Although there are numerous methodologies and tools to choose from, this dissertation focused on Scratch, Lego Mindstorms, and Blockly as example IPEs. Scratch is useful for students of all ages but has found a special niche for upper elementary to middle school students. Berkeley’s SNAP extension of Scratch and MIT’s new Scratch release (Scratch v2.0) are also appropriate for high school and college students due to their support for parameterized abstraction. Lego Mindstorms is also appropriate for upper elementary to middle school students. Environments made using Blockly target user populations ranging from elementary school students (Code.org’s Hour of Code) to high school and university students (App Inventor). These tools, when presented at the appropriate age, can spark a student’s interest in CS (Barbosa et al., 2008; Fenwick et al., 2011;
Goadrich & Rogers, 2011; Kelleher & Pausch, 2005; Mahmoud, 2011; Uludag et al., 2011; Wolber, 2011). However, due to the lack of the required dexterity, students with motor impairments are unable to use IPEs such as Scratch, Lego Mindstorms, and App Inventor.

Motorically challenged students cannot use these applications due to the dependence most IPEs have on the WIMP metaphor. Voice is a viable input for interacting with a computer, and speech can be faster than typing and can allow the user to be more productive (Hauptmann & Rudnicky, 1990; Jung et al., 2007; Martin, 1989). While multimodal interaction appears to be the most efficient method of interaction (Alsuraihi & Rigas, 2007; Bolt, 1980; Cohen et al., 1998; Hauptmann & Rudnicky, 1990; Neto et al., 2008), users with motor impairments would be unable to utilize some of the proposed tools (e.g., pen, mouse, gestures). Wobbrock et al. (2011) encourage ability-based design and support voice-driven applications if they are adaptive, perform well, and are cost effective. By focusing on a user’s ability to vocalize commands, ability-based design was practiced during the investigation and development of Myna.

The activities conducted in a typical classroom versus a special needs classroom vary significantly. Graphical programming environments are taught in typically developed classrooms (Gray et al., 2014; Marghitu et al., 2006; Marghitu et al., 2009; Miller et al., 2005), while the special needs classroom is more focused on literacy and career-oriented applications (Burgstahler, 1994; Marghitu et al., 2006; Marghitu et al., 2009; Nietupski et al., 2001). Based on hands-on activities performed in 2012 and 2013, some students with special needs are interested in learning how to program using fun, graphical environments; however, some students do not have the computer experience necessary to have an interest, which is why the literacy programs are important. Other students prefer more social or physical activities than
what a computer can provide, and others still are unable to utilize graphical programming environments due to their dependence on the WIMP metaphor.

This work presented the creation of a VUI, Myna, for a block-based IPE requiring both navigation and object manipulation. Previous work (Alsuraihi & Rigas, 2007; Begel, 2005; Bolt, 1980; Cohen et al., 1998; Dai et al., 2004 Désilets et al., 2006; Harada et al., 2009; Neto et al., 2008; Shaik et al., 2003) demonstrated that vocal navigation was successful and beneficial to users; therefore, a VUI is an appropriate approach. However, mapping a VUI to a GUI requires more than navigation. The VUI must allow the user to perform all necessary tasks (e.g., delete), which may require the creation of additional commands not native to the GUI due to the WIMP metaphor (i.e., the user would normally perform multiple steps to accomplish the task with a keyboard/mouse, but the VUI must have a vocal command that initiates the same behavior). Because the commands are not native to the application, additional code must be written to handle the appropriate behavior. This could be seen as an advantage for the vocal user (i.e., stating one vocal command versus performing three steps with a keyboard/mouse). Multiple issues (e.g., parameters, deletion, dynamic behavior, and stopping voice recognition) were encountered during the mapping process. These issues and their solutions were developed and presented in this dissertation.

Two small user studies and a user review were conducted on the overall usability of Myna. The results demonstrate that Myna is a worthwhile tool as the users were “satisfied” after using Myna, and the time to complete the programs using Myna was not significantly longer than the time taken to complete the programs via mouse/keyboard. Moreover, Myna was evaluated based on Wobbrock et al.’s (2011) ability-based design principles revealing that Myna met three of the seven requirements set forth by Wobbrock et al. (2011): Ability, Performance, and
Commodity. However, there could be improvements made to Myna to meet the Accountability, Transparency, Context, and Adaptation. Thus, changes were made to Myna including:

1. A wizard to customize the grammar – Adaptation;
2. “undo” and “start over” commands – Context;
3. Vocal command display – Transparency; and
4. Changed the system by changing the grammar for the parameter command – Accountability.

Myna can continue to be improved to better meet the ability-based design principles as can any application.

The concept of a voice-driven IPE is an excellent tool but can suffer from a static implementation strategy that is focused on a hardcoded adaptation for each IPE that is considered. It is time consuming to fully integrate voice into each new IPE. A semi-automated approach was presented that involves two tools: MynaMapper and MynaScript. MynaMapper allows a user to map the components on the screen based on the category of the various types of components (e.g., movable component, static component, command palette, editor). From the information entered by the user, the required property files are created and the grammar is generated for the Myna framework. By using MynaMapper, the user can save up to 98% of the time it would take to manually create each property file. MynaScript is a DSL that allows a user to write the necessary behaviors of the new IPE (e.g., “delete,” “drag and drop”) since there are some methods that are different among various IPEs. Through the use of MynaScript, the user need not be aware of the intricacies of Myna; the user only needs to know how the new IPE behaves.
A suggested next step for Myna would be to incorporate multiple input modalities instead of voice only. The ability to combine mouse/keyboard and eye tracking interaction with voice would allow Myna to be a more powerful tool. Another thread of inquiry would be to perform VUI testing on the resulting VUIs made with Myna by applying current GUI testing techniques. Additionally, continuing to stretch Myna’s limits by adapting it to new IPEs would be a third area for future work, particularly Code.org and any other Blockly created IPEs. Some work has already been done with Code.org, but there are many more applications on the Code.org site. Finally, and this is the most important piece of future work, more user testing should be conducted and more awareness made for the need for tools such as Myna. CS needs more developers to build assistive technology, and the best developers would be those who need the devices themselves. As the Lorax says, “Unless someone like you cares a whole awful lot, nothing is going to get better. It’s not” (Dr. Seuss, 1971).
REFERENCES


Chang, T., Yeh, T., & Miller, R. (2011, October), Associating the visual representation of user interfaces with their internal structures and metadata. *Proceedings of the 24th annual ACM symposium on User Interface Software and Technology*, Santa Barbara, CA, pp. 245-256.


Gamma, E., Helm, R., Johnson, R., Vlissides, J. (2000). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.


APPENDICES
Adding Commands and Components

Many applications will contain different components (there are three types: Movable, Movable Container, and Static) or there may be a need to add a new vocal command. Instructions for making these changes are included in the following sections.

How to Add a New Vocal Command

A vocal command is an action the user may perform, such as: “drag and drop,” “delete,” or “pause.” This is a behavior that Myna is to perform, not a block. The following steps describe the process for adding a new command to Myna.

1. Add the new vocal command (including spaces – this should be written as you would say it (numbers should be written in word form not numeric form)) to the String array ACTION_COMMANDS in CommandParser.java in the command folder.

2. Add the same vocal command to the grammar file, defaultCommands.gram. It should be written as the following on its own line. The name in brackets is essentially a variable name and should not have spaces. The text after the equal sign is the vocal command and should match exactly what was entered in CommandParser.java.

   ```
   public <newCommand> = new command;
   ```

3. Add the new method to CommandExecutor.java in the command folder. The method should have a public void signature and can be named anything that seems appropriate for the new command being added.

4. Add an if statement to the performAction method in Myna.java (default folder), which will instruct Myna what method to call if given that command. If the command requires a component (e.g., “drag and drop”), the if statement should be placed below “else if (isDoubleWord).” Otherwise, the if statement should be placed within “if (component == null).” The if statement should look like the below statement:

   ```
   else if (actionCommand.equals("delete")) {
     executor.delete(component);
   }
   ```

   “delete” in the first line is the vocal command. “executor” is the instance of CommandExecutor, and “delete” in the second line is a method call to method just created.

5. Add the new command to the first if statement in the deliverObjects method in ConcreteComponentFactory.java.

   ```
   if(componentName.equals("pause") || componentName.equals("resume") ||
   componentName.equals("okay") || componentName.equals("help") ||
   componentName.equals("done") {
     return null; }
   ```
Adding a new movable component

1. Create a property file for the block in the movableBlocks folder. The name of the file should match the vocal command used to access the block (include spaces). Below are the current properties utilized in Myna:

   ```
   name = ask and wait
   nickName = se4
   parentName = sensing
   initialLocationX = 20
   initialLocationY = 276
   isInFocus = true
   length = 190
   height = 20
   numParams = 1
   propertyLength1 = 28
   ```

   - name: the vocal command of the block and should be the name of the property file.
   - nickname: this is not currently utilized, but may be required in the future. Use any name you would like but keep it consistent throughout.
   - parentName: this is for the purposes of a dynamic grammar. In Scratch, there are menus, and each menu contains different blocks. The parentName is the menu.
   - initialLocationX: the x-coordinate (pixel) where the mouse should be directed in order to click on the block.
   - initialLocationY: the y-coordinate (pixel) where the mouse should be directed in order to click on the block.
   - isInFocus: this is not currently utilized, but may be helpful to have for future work.
   - length: the length in pixels of the block – used to determine spacing on the screen.
   - height: the height in pixels of the block – used to determine spacing on the screen.
   - numParams: this is the number of parameters included in the block.
   - parameterLength1: this represents the first parameter on the block. Some blocks have multiple parameters and some have none. If there are no parameters, delete this property. If there are multiple parameters (Scratch does not have any blocks with more than three parameters), use parameterLength1, parameterLength2, and parameterLength3 as needed. The number represents the distance from initialLocationX to where the mouse should click on the parameter slot sometimes resulting in a negative number depending on the block.

   If more properties are required, please see Step 2; otherwise, go to Step 3.

2. AbstScratchComponent.java (in the components folder) has the five basic properties (name, initialLocationX, initialLocationY, nickName, parentName). If there is a need to edit or add a new basic property, it should be added here. For additional properties such
as additional parameters, add/edit AbstMovableComponent.java. The new properties should be added to the instance variables as well as to the JSON keys and the constructor. Getters and Setters should also be created for each new property.

3. Edit defaultCommands.gram to include the new block. Because this is a movable block, add the name given to the block to both the <drag_action> and <drag_drop_action> sections. Make sure to keep a vertical line (|) between each command.

Adding a Container block such as an if statement or loop

1. Create a property file for the block in the movableBlockContainers folder. The name of the file should match the vocal command used to access the block (include spaces). Below are the current properties utilized in Myna:

```java
name = forever
nickName = c5
parentName = control
initialLocationX = 13
initialLocationY = 374
isInFocus = true
length = 71
height = 42
initialInsertHeight = 20
numParams = 0
```

- name: the vocal command of the block and should be the name of the property file.
- nickname: this is not currently utilized, but may be required in the future. Use any name you would like but keep it consistent throughout.
- parentName: this is for the purposes of a dynamic grammar. In Scratch, there are menus, and each menu contains different blocks. The parentName is the menu.
- initialLocationX: the x-coordinate (pixel) where the mouse should be directed in order to click on the block.
- initialLocationY: the y-coordinate (pixel) where the mouse should be directed in order to click on the block.
- isInFocus: this is not currently utilized, but may be helpful to have for future work.
- length: the length in pixels of the block – used to determine spacing on the screen.
- height: the height in pixels of the block – used to determine spacing on the screen.
- initialInsertHeight: this is the length from the top of the block to the point where the new block should be placed within the container.
• numParams: this is the number of parameters contained within the block.
• parameterLength1: this represents the first parameter on the block. Some blocks have multiple parameters and some have none. If there are no parameters, delete this property. If there are multiple parameters (Scratch does not have any blocks with more than three parameters), use parameterLength1, parameterLength2, and parameterLength3 as needed. The number represents the distance from initialLocationX to where the mouse should click on the parameter slot sometimes resulting in a negative number depending on the block.

If more properties are required, please see Step 2; otherwise, go to Step 3.

2. AbstScratchComponent.java (in the components folder) has the five basic properties (name, initialLocationX, initialLocationY, nickName, parentName). If there is a need to edit or add a new basic property, it should be added here. For additional properties such as additional parameters, add/edit AbstMovableComponent.java. The new properties should be added to the instance variables as well as to the JSON keys and the constructor. Getters and Setters should also be created for each new property.

3. Edit defaultCommands.gram to include the new block. Because this is a movable block, add the name given to the block to both the <drag_action> and <drag_drop_action> sections. Make sure to keep a vertical line (|) between each command.

4. If insert information needs to be added/edited (i.e., there are multiple insert points for a container), add the necessary properties to the property file, and add the same information to the BlockContainer.java file (in the components folder). Add/edit instance variables, the constructor, the load and save (JSON) methods, and getters and setters.

Adding a static element (e.g., File, Edit, or a menu name)
1. Create a property file for the block in the staticComponents folder. The name of the file should match the vocal command used to access the static component (include spaces). Below are the current properties utilized in Myna:

```java
name = file
nickName = file
parentName = scratch
initialLocationX = 231
initialLocationY = 43
```

• name: the vocal command of the component and should be the name of the property file.
• nickname: this is not currently utilized, but may be required in the future. Use any name you would like but keep it consistent throughout.
• parentName: this is for the purposes of a dynamic grammar. In Scratch, there are menus, and each menu contains different blocks. The parentName is scratch in this example because file is a parent itself.
• initialLocationX: the x-coordinate (pixel) where the mouse should be directed in order to click on the component.
• initialLocationY: the y-coordinate (pixel) where the mouse should be directed in order to click on the component.

If more properties are required, please see Step 2; otherwise, go to Step 3.

2. AbstScratchComponent.java (in the components folder) has the five basic properties (name, initialLocationX, initialLocationY, nickName, parentName). If you feel you need to edit or add a new basic property, it should be added here. For additional properties, ScratchComponent.java. The new properties should be added to the instance variables as well as the constructor. Getters and Setters should also be created for each new property.

3. Edit defaultCommands.gram to include the new component. Because this is a static component, it only needs to be placed in the <click_action> section. Make sure to keep a vertical line (|) between each command.

**Editing Property Files**
If there is a need to add or change the name of a property within a property file, the following files must be updated accordingly.
Block.java
BlockContainer.java
AbstMovableComponent.java
ConcreteComponentFactory.java
Make sure that the new property is named consistently throughout the different files, particularly for the JSON keys.

**Drop Location**
By default, the drop location for the first block is (300,200). Myna also assumes that blocks will be placed vertically. Below are instructions for changing the default drop location in addition to changing the subsequent block placement.

*Changing the initial drop location within the program*
In ScriptsState.java, the default drop location for starting the program is (300,200). To edit this location, these values will have to be changed in two locations within the ScriptsState.java.
public ScriptsState() {
    movableComponents = new Vector<AbstScratchComponent>();
    components = new LinkedHashMap<Integer, AbstScratchComponent>();
    dropX = 300;
    dropY = 200;
}

private ScriptsState(Vector<AbstScratchComponent> movableComponents,
                      Map<Integer, AbstScratchComponent> components,
                      int dropX, int dropY) {
    this.movableComponents = movableComponents;
    this.components = components;
    this.dropX = dropX;
    this.dropY = dropY;
}

private Vector<AbstScratchComponent> movableComponents = new
    Vector<AbstScratchComponent>();
private Map<Integer, AbstScratchComponent> components = new
    LinkedHashMap<Integer, AbstScratchComponent>();
private int dropX = 300;
private int dropY = 200;

Changing the subsequent drop location of a component
In Myna, the subsequent drop location is after the previously dropped block, and blocks are
placed vertically since Scratch v1.4 operates in this fashion. Below are instructions for changing
the block placement to horizontal block placement. Note that drag and drop, drop before, drop
after, and drop in will all need to be altered (these are methods found in the
CommandExecutor.java file). The documentation below covers drag and drop only, but the other
methods will be changed in a similar fashion.

From CommandExecutor.java
public void dragAndDrop(AbstScratchComponent comp) {  
    1. Determine the current xy location of the component to be moved
       int currentX = comp.getX();
       int currentY = comp.getY();
    2. Get the current ScriptsState, which stores the movable components as
       they are added to the program
       ScriptsState scriptsState = AppState.getCurrentScriptsState();
    3. Get the current drop point, the point currently recorded as the next
       location to drop a block
       int tempCurrentDropPointX = scriptsState.getDropX();
       int tempCurrentDropPointY = scriptsState.getDropY();
4. Move the mouse to the current location of the component and grab the component

```java
robot.mouseMove(currentX, currentY);
robotMousePress();
```

5. Move the mouse (dragging the component) to the drop point

```java
robot.mouseMove(tempCurrentDropPointX, tempCurrentDropPointY);
```

6. Update the component’s xy location

```java
//change component’s current location
comp.setX(tempCurrentDropPointX);
comp.setY(tempCurrentDropPointY);
```

7. Add the component to the current ScriptsState

```java
//add the component at the end of the Sprite State Vector
AppState.getCurrentScriptsState().getMovableComponents().add(comp);
```

8. Grab the component’s height and adjust the current drop point by the component’s height. This is all the current drop point needs to be adjusted in this example since Scratch connects by touching block to block.

```java
int compHeight = comp.getHeight();
this.moveCurrentDropPoint(compHeight);
```

9. Release the Cracken...er, mouse.

```java
robotMouseRelease();
```

From CommandExecutor.java

```java
private void moveCurrentDropPoint(int componentHeight) {

1. Grab the current ScriptsState

ScriptsState scriptsState = AppState.getCurrentScriptsState();

2. Get the current drop xy location adding the componentHeight to the y location since we would want to move the drop point further down the screen

    ```java
    int currentDropX = scriptsState.getDropX();
    int currentDropY = scriptsState.getDropY() + componentHeight;
    ```

3. Update the ScriptsState’s drop point to the new location

    ```java
    scriptsState.setDropX(currentDropX);
    scriptsState.setDropY(currentDropY);
    ```
```
To adjust from vertical to horizontal placement, the formulas for `int currentDropX` and `int currentDropY` will need to be reversed. Also, if the blocks are not meant to be dropped so that each block is touching the previous block, the default space should be added.

**Creating a .jar File for Myna**
Below are the steps for creating an executable .jar file for Myna:

- Right-click project in Eclipse
- Select Export
- Under Java
  - Select Runnable JAR file
  - Click Next
- In the Runnable JAR File Export Form
  - Launch configuration should be set to Myna
  - Choose your Export destination
  - Choose Extract required libraries into generated JAR
  - Click Finish
APPENDIX B
Pilot Study Testing Instructions

Instructions:
There are three exercises described below. Please perform each exercise in Scratch (using the mouse and keyboard). After completing the three exercises in Scratch, please perform each exercise in Myna (using only voice). Your performance will be observed to document the time taken to complete each exercise as well as how Myna performs. At the end of the experiment, you will be asked to complete an experience survey regarding how you feel about using Myna.

Exercise #1:
1. To see a list of the vocal commands, say, “help.”
2. To begin the program, we want to use the “when clicked” control block. This means that when the green flag is clicked, the program will run.
3. Next, we want to put the pen down in order to draw a line behind the Scratch Cat.
4. The Cat should move steps using the default number of steps.
5. The Cat should say (hello) for seconds.
6. The Cat should go to x y using the default xy location.
7. Go to the Control menu.
8. Scroll down (scroll command down) and add “stop all”.
9. Finally, execute (play) the program.

Exercise #2:
1. To begin the program, we want to use the “when clicked” control block. This means that when the green flag is clicked, the program will run.
2. The Cat should go to x y using the default xy location.
3. Change the parameters for go to x y (“set property” x “at” y) to x = 5 leave as the default value.
4. The Cat should move steps using the default number of steps.
5. Change the parameter for move steps to 20.
6. We want to put the pen down in order to draw a line behind the Scratch Cat, but this should be dropped after when clicked.
7. Finally, execute (play) the program.

Exercise #3:
1. To begin the program, we want to use the “when clicked” control block. This means that when the green flag is clicked, the program will run.
2. We want to put the pen down in order to draw a line behind the Scratch Cat.
3. The Cat should turn degrees clockwise.
4. The Cat should move steps using the default number of steps.
5. The Cat should go to the default location.
6. Delete go to.
7. Finally, execute (play) the program
APPENDIX C
Pilot Study Observation Instrument

This instrument will be used to record each observation during user testing.

Participant ID: ________________________________________
Date: ________________________________________
Observer: ___________________________________________

1. Number of participant errors: _____
   a. Participant stated the incorrect vocal command (“Drag” vs. “Drag and Drop”): _____
   b. Participant stated the incorrect Scratch command (“Go to x” vs. “Go to xy”): _____

2. Number of Myna errors: _____
   a. Voice recognition was inaccurate (Participant said the correct commands, but Myna
      reacted incorrectly): _____
   b. The xy-location on the screen was incorrect: _____

3. Time to complete experiment using mouse/keyboard: _______

4. Time to complete experiment using voice: _______

5. Note any issues the participant experiences.
APPENDIX D
Pilot Study Myna Experience Survey

Myna Experience Survey (Pilot Study)
The purpose of this survey is to evaluate the effectiveness of programming in Scratch using voice. Your feedback is greatly appreciated.

General Commands
The following questions concern the functionality of the vocal commands, which are the key commands used to build the program. Please circle the number that best represents your response to each question.

1. I feel the vocal commands in Myna are:
   1. Predictable
   2. Somewhat predictable
   3. Neither predictable nor unpredictable
   4. Somewhat unpredictable
   5. Unpredictable

2. I feel the vocal commands in Myna perform the actions that I expect:
   1. Always
   2. Most of the time
   3. Half of the time
   4. Some of the time
   5. Never

3. I feel the commands in Myna are relevant to the action being performed:
   1. Always
   2. Most of the time
   3. Half of the time
   4. Some of the time
   5. Never

Specific Features
The following questions concern the way specific features work within Myna. Please circle the number that best represents your response to each question.

1. I find the screen resolution in Myna to be:
   1. Frustrating
   2. Somewhat frustrating
   3. Neither frustrating nor pleasing
   4. Somewhat pleasing
   5. Pleasing

2. The “Delete” command works as expected in Myna:
   1. Never
   2. Occasionally
   3. Half of the time
   4. Most of the time
   5. Always

3. Parameters (as used to provide information in a block to perform a specific operation) are easy to edit in Myna:
   1. Never
   2. Occasionally
   3. Half of the time
   4. Most of the time
   5. Always
4. It is easy to pause Myna to stop listening to my voice:
   1 2 3 4 5
   Never Occasionally Half of the time Most of the time Always

5. It is easy to resume Myna to listen to my voice:
   1 2 3 4 5
   Never Occasionally Half of the time Most of the time Always

Usability
The following questions concern your experience using Myna. Please circle the number that best represents your response to each question.

1. I think error messages (messages indicating there is a problem) in Myna are:
   1 2 3 4 5
   Unhelpful Somewhat neither unhelpful nor helpful Somewhat helpful

2. I think help messages (messages providing usage tips) in Myna are:
   1 2 3 4 5
   Unhelpful Somewhat neither unhelpful nor helpful Somewhat helpful

3. I feel tasks can be performed in a straight-forward manner in Myna.
   1 2 3 4 5
   Never Occasionally Half of the time Most of the time Always

4. I perceive the use of terms throughout Myna to be:
   1 2 3 4 5
   Inconsistent Somewhat neither inconsistent nor consistent Somewhat consistent

5. I think Myna keeps me informed about what it is doing.
   1 2 3 4 5
   Never Occasionally Half of the time Most of the time Always

6. I think supplemental reference materials for Myna would be:
   1 2 3 4 5
   Unhelpful Somewhat neither unhelpful nor helpful Somewhat helpful
Prior Knowledge

The following questions concern your prior knowledge of Scratch and Myna. Please circle the number that best represents your response to each question.

1. My knowledge of Scratch is:
   1  2  3  4  5
   Very High Somewhat high Neither high nor low Somewhat low Low

2. I have used Scratch:
   1  2  3  4  5
   Never Once Two or three times Three to five times More than five times

3. My knowledge of Myna is:
   1  2  3  4  5
   Very High Somewhat high Neither high nor low Somewhat low Low

4. Prior to this study, I have used Myna:
   1  2  3  4  5
   Never Once Two or three times Three to five times More than five times

Learning Myna

The following questions concern the degree to which Myna is easy to learn.

1. I perceive the degree to which Myna is easy to learn as:
   1  2  3  4  5
   Very difficult Somewhat difficult Neither difficult nor easy Somewhat easy Very easy

2. I find exploring new features by trial and error in Myna to be:
   1  2  3  4  5
   Very difficult Somewhat difficult Neither difficult nor easy Somewhat easy Very easy

3. I find remembering names and uses of commands in Myna to be:
   1  2  3  4  5
   Very difficult Somewhat difficult Neither difficult nor easy Somewhat easy Very easy
Overall Impression

The following questions concern your overall impressions of Myna. Please circle the number that best represents your response to each question.

1. After using Myna, I am:

   1           2           3           4           5
   Very dissatisfied  Dissatisfied  Neither dissatisfied nor satisfied  Satisfied  Very satisfied

2. Using Myna made me:

   1           2           3           4           5
   Frustrated  Somewhat frustrated  Neither frustrated nor contented  Somewhat contented  Contented

3. After using Myna, I feel:

   1           2           3           4           5
   Terrible  Unhappy  Neither terrible nor delighted  Happy  Delighted

Demographic Information

All of the below questions concern your demographic information.

1. Age:

   20-24  25-29  30-34  35-39  40+

2. Gender:

   Male  Female

3. Race:

   Asian, Asian American, or Pacific Islander
   Black or African American
   Native American
   South, Latin, or Central American or other Hispanic
   White or Caucasian
   Other: ____________________________

4. Highest Level of Education:

   Undergraduate  Some Graduate Coursework
   Degree  Graduate Degree
### APPENDIX E
Pilot Study Myna Satisfaction Survey Results

<table>
<thead>
<tr>
<th>Question</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
</tr>
</thead>
</table>
| **General Commands**  
I feel the vocal commands in Myna are                                   | 4   | 5   | 5   | 4   | 4   |
| I feel the vocal commands in Myna perform the actions that I expect     | 3   | 4   | 4   | 3   | 4   |
| I feel the commands in Myna are relevant to the action being performed  | 4   | 5   | 4   | 4   | 5   |
| **Specific Features**  
I find the screen resolution in Myna to be                              | 4   | 5   | 5   | 5   | 3   |
| The "Delete" command works as expected in Myna                          | 2   | 4   | 5   | 2   | 5   |
| Parameters are easy to edit in Myna                                     | 3   | 1   | 3   | 2   | 4   |
| It is easy to pause Myna to stop listening to my voice                  | 5   | 4   | 5   | 5   | 5   |
| It is easy to resume Myna to listen to my voice                         | 5   | 5   | 5   | 5   | 5   |
| **Usability**                                                          |     |     |     |     |     |
| I think error messages in Myna are                                      | 1   | 4   | 5   | 2   | NA  |
| I think help messages in Myna are                                       | 5   | 5   | 4   | 4   | NA  |
| I feel tasks can be performed in a straight-forward manner in Myna      | 3   | 4   | 4   | 3   | 4   |
| I perceive the use of terms throughout Myna to be                      | 4   | 5   | 4   | 5   | 5   |
| I think Myna keeps me informed about what it is doing                   | 1   | 3   | 3   | 3   | 4   |
| I think supplemental reference materials for Myna would be              | 4   | 4   | 5   | 4   | 4   |
| **Prior Knowledge**                                                     |     |     |     |     |     |
| My knowledge of Scratch is                                              | 2   | 1   | 1   | 1   | 1   |
| I have used Scratch                                                     | 2   | 1   | 1   | 1   | 1   |
| My knowledge of Myna is                                                 | 2   | 1   | 1   | 1   | 1   |
| Prior to this study, I have used Myna                                   | 1   | 1   | 1   | 1   | 1   |
| **Learning Myna**                                                       |     |     |     |     |     |
| I perceive the degree to which Myna is easy to learn as                 | 3   | 4   | 4   | 4   | 5   |
| I find exploring new features by trial and error in Myna to be          | 1   | 4   | 4   | 3   | 5   |
| I find remembering names and uses of commands in Myna to be             | 3   | 4   | 3   | 4   | 4   |
| **Overall Impression**                                                  |     |     |     |     |     |
| After using Myna, I am                                                  | 3   | 4   | 4   | 3   | 4   |
| Using Myna made me                                                      | 2   | 3   | 4   | 3   | 3   |
| After using Myna, I feel                                               | 3   | 4   | 4   | 4   | 3   |
| **Demographic Information**                                             |     |     |     |     |     |
| Gender                                                                  | Male | Male | Male | Male | Male |
| Race                                                                    | White | White | Asian | Asian | White |
| Native Speaker                                                          | Yes | Yes | No | No | Yes |
| Highest level of education                                              | Grad | Grad | Grad | Grad | Grad |

163
APPENDIX F
UCP Study Observation Instrument

This instrument will be used to record each observation during user testing.

Participant ID: ________________________________________
Date: _______________________________________________
Observer: ___________________________________________

1. Number of participant errors: _____
   a. Participant stated the incorrect vocal command (“Drag” vs. “Drag and Drop”): _____
   b. Participant stated the incorrect Scratch command (“Go to x” vs. “Go to xy”): _____

2. Number of Myna errors: _____
   a. Voice recognition was inaccurate (Participant said the correct commands, but Myna reacted incorrectly): _____
   b. The xy-location on the screen was incorrect: _____

3. Time to complete experiment using mouse/keyboard: _______

4. Time to complete experiment using voice: _______

5. Note any issues the participant experiences.
APPENDIX G
UCP Study Satisfaction Survey

Myna Satisfaction Survey

This instrument is to be answered by the participant of the UCP study but completed by the observer.

Participant ID: ________________________________________

Date: _______________________________________________

Observer: ____________________________________________

1. Which face describes how you felt using Myna?

[Five smiley faces with varying expressions]

2. Did you enjoy creating your own animations with Myna?
   Yes   No

3. Would you use Myna again?
   Yes   No

4. Is there anything else you would like to say about Myna?