121st ASEE Annual Conference & Exposition
Indianapolis, IN
June 15-18, 2014
Paper ID #9206

Teaching Modern Object-Oriented Programming to the Blind: An Instructor and Student Experience

Dr. Charles B. Owen, Michigan State University
Sarah Coburn, Michigan State University
Ms. Jordyn Castor

Teaching Modern Object-Oriented Programming to the Blind: An Instructor and Student Experience

Abstract

Blind computer science students face significant challenges in modern curricula. Computer programming has seen extensive growth of visual tools and environments. Most end-user development is for visual environments, from graphical user interfaces (GUIs) to web pages. The tools of choice have become highly visual integrated development environments (IDE). Visual modeling languages such as Unified Modeling Language (UML) are a major element of program understanding and design. All of these tools and methods are problematic for a blind student. This paper describes the challenges we faced as both the instructor and student as we adapted tools, material, and assignments in an object-oriented programming course.

1 Introduction

In the fall of 2013, Jordyn, a co-author of this paper, enrolled in an intermediate-level course in object-oriented programming and modern development methodologies. Jordyn is very unique in Computer Science programs because she is blind.[1] Jordyn is the first blind student I have had in a course. Sadly, the experience has been that blind students have largely not proceeded beyond the CS1 course in the past, so Jordyn is indeed unique. As we began to examine the course requirements, we faced considerable challenges. This paper describes some of those challenges, how we overcame them, where we still struggle, and how it was possible for her to succeed in this course. Since we were able to solve many of the problems she faced in the course, we feel it is important to share these solutions, so other students can also have the chance to succeed in Computer Science programs.

It was a shock to discover how limited the tools are for blind students in Computer Science. There have been projects aimed at attracting blind students and, indeed, it was one of those programs that got Jordyn excited about the major.[2] However, there have been fewer efforts to provide support for students once they are in a curriculum. Several projects have created custom languages that are designed for blind and limited-vision users, such as Quorum[3] and Audio Programming Language (APL).[4] However, few projects have addressed the problem of students succeeding in programs where they must use languages common in curricula, such as Java, C++, and Python. Existing tools generally follow the pattern of adapting a visual tool such as NetBeans or Eclipse by tacking on accessibility that speaks the visual elements of the interface. Accessibility support in these tools is often incomplete and buggy. For example, Sodbeans adapts some NetBeans components with accessibility, but does not adapt the debugger.[5] Even more problematic are visual development tools such as UML diagrams that have no obvious textual equivalent. We have been able to create some simple system extensions to make Sodbeans and GDB, the GNU project debugger we utilize in the course, work better together,

and we have developed textual, tactile, and audible representations of UML diagrams that are accessible to a blind student.

This course represented a unique combination of an instructor who is an expert in multimedia and user interfaces with a student who is highly motivated to learn and succeed. We were able to work together throughout the semester, trying different approaches, learning what worked and discarding what did not. As problems arose, we addressed them through adapting existing tools when possible, creating smaller fixes, and creating new tools better adapted to her needs. We have, in the process, learned a lot about how to make an effective aural interface.

2 Structure of the Course

The course is called Object-Oriented Programming and is meant to be the transition from simple Python and C++ programs to larger systems, where system design is important and programs span many classes. The course includes modern design methods and UML diagrams as an important element. The course also introduces Design Patterns, utilizing many of these in projects.[6] There is an introduction to practical concurrency and multithreading.

The course has many structural elements. There are two lectures per week. During lectures, students are given a worksheet referred to as a “ToeTipper” that they complete during class. This allows students to gain experience solving problems in class and breaks up the lecture. There are assignments referred to as “Step Assignments”. A step assignment is a combination of reading, tutorial, code literacy, and individual tasks. There are two larger projects, a group project and an individual project, though the individual project does include a group design component. There are several additional “Design Assignments” that require students to solve a design problem and submit UML diagrams. The design assignments are subject to an anonymous peer review system.

The course is taught in the C++ programming language and students enter the course with one previous course in that language. The assignments are all graphical user interface applications using the wxWidgets class library.[7] Visual Paradigm is used as a UML editor.[8] NetBeans is used as an integrated development environment. Students work on the Linux platform.

3 The Student

Jordyn became interested in computers as a career from an early age. She began using a computer in the second grade and it was obvious to her and her family that technology would play an important role in everyday life. While in the seventh grade, she attended the National Federation of the Blind *Youth Slam*, where she wrote her first C# computer program, a chat bot that could communicate through Windows Live messenger.[2] The chat bot was able to perform actions like checking the weather, looking up dictionary definitions, and playing Simon Says. She realized she could program a computer, that the methods for instructing a computer were by

no means out of her reach, and became extremely interested in computer science. She hopes that following that career path will enable her to help others with disabilities.

Jordyn completed the CS1 course in the Python programming language and the CS2 course in C++ at Michigan State University. For those courses, she used the VIM editor.[9] She considered other options such as Emacspeak, but found them difficult to install and use. She uses screen readers built into the OS-X operating system and Orca screen reader on Linux platforms. Working with source code using a screen reader is challenging. Sighted users are able to scan both content and structure on demand very quickly. The do not have to carry much of the design in their head. Hearing is not directed, so she must use keyboard controls to move through code as it is read by the screen reader. Compared to a sighted user, the process of directing focus is considerably slower.

4 Challenges and Solutions

As we entered the semester, there was some concern about visual diagrams, but it was assumed that other tools would be sufficient for the course. A version of NetBeans, the IDE the course uses, had already been adapted for blind users, and most tools in the Linux world have command line versions. Adapting graphical material is a task that the Resource Center for Persons with Disabilities (RCPD) had considerable experience with and they had converted diagrams in math courses in the past. However, it was quickly clear that most of these problems were much farther from being solved than we assumed.

4.1 Available Tools

The student used Vinux running under a VMware virtual machine on a MacBook Pro running OS-X Mavericks. Vinux is an Ubuntu-derived variation of Linux specifically configured for the needs of blind and partially sighted users.[10] Vinux includes the Orca screen reader, which provides verbalizations of user interface elements.[11] The accessibility of the Gnome desktop is adequate and she is able to operate and use the system moderately efficiently. She has a Focus 40 Blue Braille display, but did not find it useful for this course.

The first choice that needed to be made was what programming environments to use. In previous courses, she had used VIM, which is moderately accessible. But, this course requires management of much larger projects including many classes and functions, a task more suited to an integrated development environment. There are a variety of integrated development environments available for Linux that have different degrees of accessibility. The course utilizes NetBeans as the standard IDE and there is variation of NetBeans called Sodbeans that provides some accessibility.[5] Sodbeans was created primarily as a vehicle for using the Quorum programming language, a computer language specifically designed to introduce blind students to programming. Sodbeans extends NetBeans with spoken output in the editor and many of the program management components. It is not a complete accessibility solution, as will be

discussed, but it did enable management, editing, and building of larger projects and was compatible with projects other students created in the course. There was an issue that debugging in Sodbeans is not accessible, as will be discussed in Section 4.4.

4.2 Visual Diagramming

The course teaches program design using Unified Modeling Language (UML) diagrams. Specifically, the class, object, and state diagrams are covered and students are expected to provide designs in all three. Much of the course material includes UML diagrams to convey program structure. An example UML class diagram from the course is shown in Figure 1. UML class diagrams convey the attributes and operations for a class, inheritance and nesting relationships, and the associations between classes. They are a visual way of representing the structure of an object-oriented system.

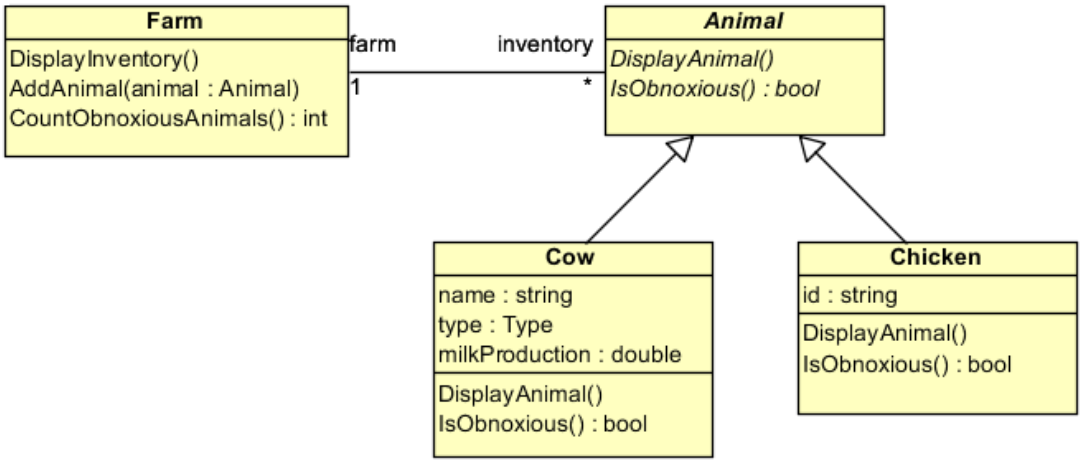

Figure 1 - Example UML Class Diagram

Of course, she could not see a UML diagram. Therefore, there were several major problems that had to be overcome: a) helping her to understand what the diagrams normally convey visually, so she can understand what the diagrams teach about program structure, b) providing a way to represent a UML diagram that she could both read and produce, and c) providing a way for her to participate in the diagram assignments and, in particular, the peer review process.

The natural idea for conveying UML diagrams was to convert them to raised-dot diagrams, where the structure is indicated by raised dots on paper and the textual elements are indicated in Braille. RCPD has a Tiger Braille printer capable of converting graphics to raised dot and Braille diagrams and did convert several diagrams.[12] However, this was not an effective approach for several reasons. The printer is not able to read the diagrams produced by Visual Paradigm. So, the diagrams had to be manually reproduced by a sighted user in a format acceptable to the printer. Since Braille letters are nominally 10mm tall and 6.2mm wide, they are considerably larger than normal text. The widest line in Figure 1 is 30 characters wide, or 18.6cm wide.

Maintaining scale on the diagram would require a diagram 60cm wide, considerably larger than the Braille printer could accommodate. And, this was a relatively small class diagram with only four classes. Later diagrams in the course included as many as 18 classes. The solution used by RCPD was to present the diagram as blocks with a key in each block and a separate page that listed the components of the block. When combined with roles and multiplicities on edges, the contents of the diagram was effectively scattered over many pages and became very difficult to comprehend. She was not able to gain much understanding of the diagrams from the raised-dot representations.

Attempts to verbally describe the structure of a UML diagram were not effective. Conveying physical relationships among graphical elements verbally is difficult to do and there was no common frame of reference.

A solution that worked reasonably well for conveying the *concept* of the diagrams and their meaning was to create a way that she could physically feel and manipulate a diagram. Figure 2 illustrates the solution we utilized. A cork bulletin board was placed on a desktop. Post-it notes were created to represent classes. The elements of the class were added to the Post-it notes using a slate and stylus (portable Braille embosser). This made it possible for her to feel the relative placement of the classes in the space.

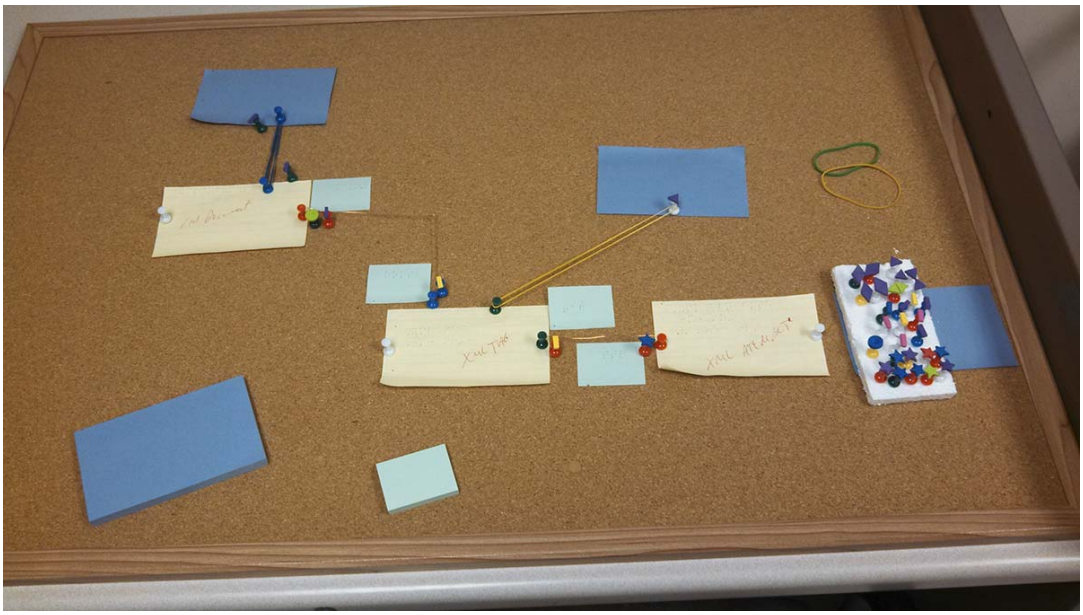

Figure 2 - Physical representation of UML diagrams

Regular push-pins were inserted into the board on the edges of classes. The edges between classes could then be represented by either wire or rubber bands. Wire allowed the instructor to create links that reflected those on the drawing, with bends and a single, sturdy line. Rubber bands allowed the student to modify the diagram herself. Roles were indicated on associations using smaller Post-it notes, also with Braille.

A set of foam symbols was found at a local craft store that included several symbols close to the UML symbols used on edges. Figure 3 shows some of these symbols and their corresponding UML element. The zero, one, and star symbols represented the most common multiplicities. Inheritance was indicated using a triangle. Composition was indicated with the two triangles that formed the closed diamond structure.

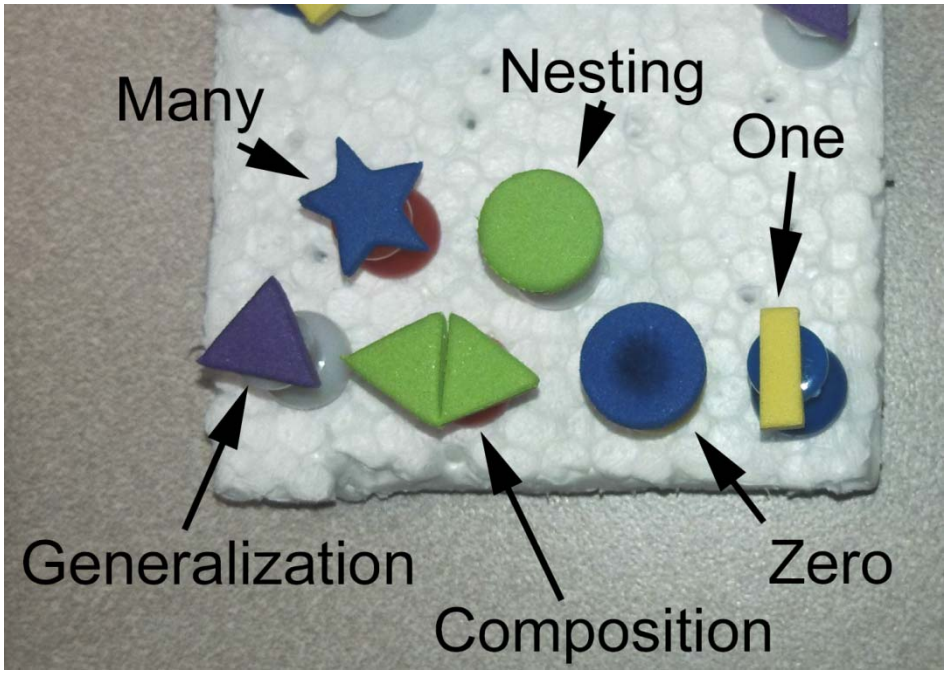

Figure 3 - Foam UML symbols on push pins

This approach had the advantage that she could not only comprehend a diagram that had been made through tactile manipulation, she could create diagrams herself. The pins were easy to manipulate and a rubber band created associations or generalizations. The approach was used for both class and object diagrams in the course.

The semester following completion of the object-oriented course, Jordyn enrolled in the data structures course taught by another instructor. She experienced a similar difficulty understanding the concept of trees and graphs. Again, the bulletin-board approach provided invaluable for helping her to understand the meaning of those structures.

For assignments and presentation of content after she reached an understanding of the diagram structure, we devised a textual representation of the diagrams. The syntax consisted of the word “class” and the class name and any inheritance or nesting, followed by the attributes and operations. Associations where then listed as “association”, “aggregation”, or “composition” followed by the first class, role, and multiplicity, then the second class, role, and multiplicity. Italics, which is used to indicate abstract classes and operations, is indicated using slashes. As an example, a subset of the diagram in Figure 1 would be:

```
class Farm
DisplayInventory()
AddAnimal(animal : Animal)
CountObnoxiousAnimals() : int
class /Animal/
/DisplayAnimal()/
/IsObnoxious() : bool/
association Farm farm 1 Animal inventory *
```

A similar format was devised for object and state diagrams. All diagrams in the course prior to the introduction of Audible Browser were translated into this format. Her solutions to the design assignments were translated to visual diagrams by a volunteer, who also translated other student submissions assigned for review to the textual format. This format was also used for the exams.

Late in the semester, an experimental program, Audible Browser, for navigating UML diagrams audibly was devised. Audible Browser is discussed in its own section.

4.3 Visual and GUI Assignments

The course uses a group of structured assignments called "Step Assignments" consisting of tutorials, code reading, and individual work. These lead to group assignments. These assignments are GUI-based, using the wxWidgets interface library, and highly visual in nature, building programs such as an aquarium simulator, animation system, and elevator simulator.

The earliest solution for this problem was for her to work with a sighted student who could serve as eyes on the project and use the mouse for manipulations. This was a less than ideal solution, requiring coordination of schedules and limiting the time she could work on the project. Providing an alternative assignment was not a good solution, since the Step assignments are large, carefully designed to exercise the elements of the course, and are based on a combination of code reading in a larger design and problem solving. So, we began to examine how we could make it possible for her to complete the assignments and test the functionality of an interactive GUI application without being able to see the screen.

Over several iterations, I created CSpeech, a drop-in component she could easily add to an assignment. CSpeech is being made available as an open-source component (http://metlab.cse.msu.edu/blind.php). CSpeech made it possible for her to add speech to an application, adding commands that worked like simple cout statements. CSpeech uses the espeak library for voice synthesis. Once added to the application, she could make the program speak at any point using a command like this:

```
speak << “The current position is ” << x << “, ” << y << flush;
```

The class worked like cout with additional functions to support output of wxWidgets types like wxSize and wxRect. The component also supports output of tone sequences using this syntax:

```
speak.Tone(0.15, 1000, 0.5);
```

This statement plays a 1000Hz tone for 0.5 seconds at an amplitude of 0.15. Tones provide a sound cue that is faster than speech and can convey an additional variable through variations in pitch.

CSpeech installs an accessible menu *Speech* when it is used. The menu includes options to list the current windows on the screen including child windows, speak the window the mouse is hovering over, and speak the x,y location in a window that the mouse is hovering over. It also includes a command to empty and flush the current speech queue, a menu option affectionately referred to as *Shut Up*. All of the menu options are available through accelerator keys. Most blind computer users are very experienced in the use of keystrokes for menu access and tabbing through fields in dialog boxes.

Whereas a conventional GUI application will draw the screen. Jordyn would add statements to speak the status of the screen. For the aquarium simulation, the program would speak the location and type of the fish on the screen. A modification of the mouse move event handler allowed the program to speak when the mouse hovered over a fish, so she could grab it and move it. It did require a search behavior, but speaking the mouse location did help in the process. She was able to successfully complete the assignment.

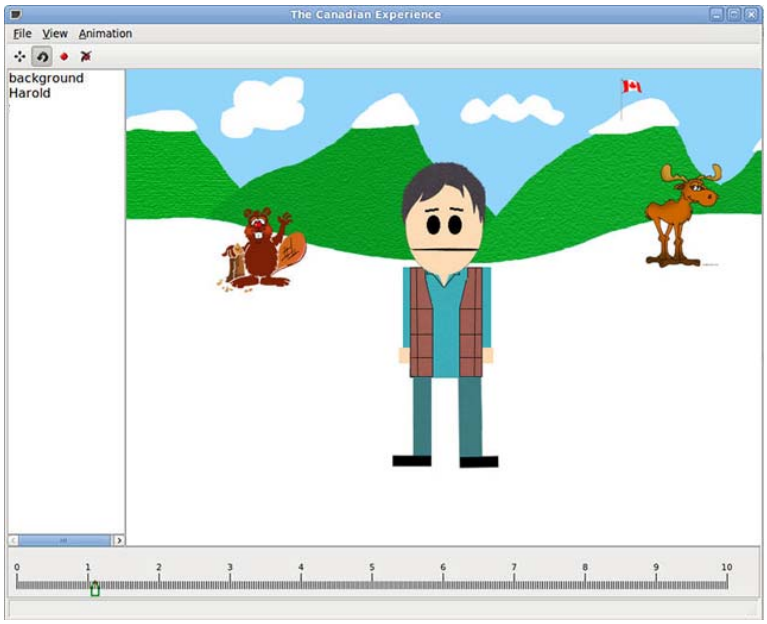

Figure 4 - Animated character step assignment

A later assignment, illustrated in Figure 4 included an articulated animated character, where students create a timeline and create keyframe animation for the rotations and positions of joints. This was more problematic to speak, since there are a larger number of components in a character. For this task, we used tones to play the current angle or position. Each redraw resulted in a tonal sequence. Animation operation was indicated by changes in pitch. Some modifications of the assignments were made for accessibility. Manipulating the timeline is normally done using the mouse. She completed the same activity using keyboard commands.

A common problem was a cacophony of voices. The speech generation from CSpeech is independent of Orca, the screen reader. Speech in Sodbeans is also independent. So, it was not uncommon that the systems would attempt to speak simultaneously. A common solution was to add a short delay using speak.Delay(1.0) to allow Orca to complete reading a window name. The ability to cancel the current speech in both systems was found to be very important.

Surprisingly enough, the accessibility features in Linux do not make it easy to just send a string of text to Orca to speak. The accessibility libraries, such as AT-SPI,[13] are designed to allow the screen reader to access text in on-screen controls. It is a pull model where screen readers request accessibility information. There is also no reasonable means to determine the duration of an utterance or intersperse other audible content with the speech.

4.4 Debugging

A significant problem was debugging. While Sodbeans has a debugger specifically for the Quorum language, they did not extend the program to support debugging of C++ or Java applications, so those parts of the user interface remained inaccessible. The solution was to use GDB directly from the command line. While the interface is more cumbersome, it is accessible, since it is all text. It is sometimes excessively verbose, dumping considerably more information than necessary at the time. Sighted users can easily scan over these dumps, but a blind user has to sit through considerable reading.

A problem with using gdb from the command line was that a GUI application seizes focus when it starts, and the screen readers only read content from a screen that currently has focus. This was particularly a problem when the program hit a breakpoint or segfaulted. The focus either remained on the program or would transfer to Sodbeans or some other program. Consequently, the gdb output of the breakpoint or fault would not be spoken by the screen reader. The effect often was that the program failed silently.

A solution to this problem was to add this content to the .gdbinit file:

```
python
import os
def stop_handler (event):
      os.system("espeak gdb")
```

```
gdb.events.stop.connect (stop_handler)
```

Whenever gdb gained control, the stop_handler function would run. That function used espeak to say “gdb.”, providing an audible indication that gdb had gained control. The student could then cycle focus to the terminal window hosting gdb.

5 Audible Browser

Comprehending the physical layout of UML class and state diagrams can be very difficult if you cannot see them. The cork board layout provided a means to convey a diagram and for Jordyn to learn to construct a diagram. However, it is too cumbersome to convey all of the examples in the course and problems on exams. The textual representation does convey the essence of the diagrams, but is not easy to scan.

Eye tracking studies using UML diagrams have shown that the eye predominantly moves among the nodes of the diagram (classes, objects, or states), fixating on nodes.[14] There are additional fixations on the ends of the links (associations, transitions, and generalization, for example). Users comprehend UML diagrams by examining the major items and the relationships between them. This makes a textual representation less effective, since it sequentially presents things like classes in an arbitrary order. While it may be possible to choose a better order, it is not going to be easy for a user to traverse paths through a class or state diagram in this way.

There have been attempts to provide audible representations of drawings. Microsoft’s InkAnalysis attempts to understand an arbitrary diagram and allow a user to navigate between items quickly[15]. Others have used tactile interfaces to support feeling graphics, but these require specialized hardware. However, the problem for UML diagrams is not recognizing that there are rectangles, but recognizing that there are classes or states and connections among them and being able to rapidly follow the connections so as to comprehend the structure or sequencing.

As an experiment, we created Audible Browser. A full description of Audible Browser is the subject of a future paper. We summarize the basic functionality here. Audible Browser will be made available as an Open Source project in the near future.

Audible Browser reads XMI 2.1 files and currently supports Class and State diagrams. We specifically are using diagrams produced by Visual Paradigm. This discussion will focus on class diagrams. The presentation of state diagrams is similar. The program presents the nodes in a diagram in a *tab group*. A tab group is a set of common items that can be read using the up and down arrow keys. The tab groups for a class are the name, attributes, operations, and associations. The name tab group initially speaks the class name and any inheritance or nesting. This is a summary of the class. The arrow keys allow the user to traverse through a list that includes any class this class is nested in, any classes nested in this class, classes this class inherits

and classes that inherit this class. Hitting return on any of these lines allows the user to move directly to the associated class.

The next tab stop presents the class attributes. These are spoken in the same form they appear in a normal UML class diagram. For example, the attribute x: int=0 is spoken as “x colon int initially zero”. The left and right arrow keys allow the user to move from letter to letter, speaking the letters. This prevents ambiguity if the speech generation is not clear.

The next tab group allows the user to examine the operations of the class. The function func(x:int, y:int) : string is read in the form: “func parameters x colon int y colon int returns string”. As before, the left and right arrow keys read out the letters of the actual UML representation one at a time.

The final tab group lists associations, aggregations, and compositions. In each case, the roles and multiplicities are read with the association. As an example: “association diagram 1 to state 0 dot dot asterisk”.

To convey the structure of a diagram, Audible Browser plays tones for each node. These tones are referred to as *stars* and are in a *constellation*. Each star is presented as a tone. The left-right placement represents the x position in the diagram and the pitch represents the y position. 3D audio is only effective for left-right placement of content. 3D audio is not effective for vertical placement for general users. Nodes that are higher on the page are presented with a higher pitch. Mansur, Blattner, and Joy found that humans have a natural tendency to interpret higher frequencies as vertically higher in space.[16] The name of the node can also be optionally spoken when the constellation is presented. This provides a way to quickly convey the relative placement of the nodes around the user.

The user can navigate to other nodes using the mouse. All mouse movement is relative to when the movement begins, so there is no absolute placement and the mouse does not correspond to screen locations. Technically, the mouse is re-centered when it is released, so there is always space for movement in all directions and it cannot be clicked outside the window and accidentally inflict a context change. When moving, the left-right placement changes to tell the user when items move to the left and right. The pitch for items stays the same. A background tone is played as the constellation is continuously presented when moving. The tone provides a reference of the current y location and varies as the user moves up and down. Humans are very poor at recognizing absolute pitches, so pitch is not a good representation of absolute position. However, humans are good at recognizing differences if pitch, so the pitch is a powerful tool for representing relative placement.[17] Traversing to a node consists of moving it so the sound is centered left and right and the motion tone matches the pitch. As nodes are neared, this status is spoken as well. When the mouse button is released, Audible Browser snaps to the nearest node.

6 Conclusion

Making an object-oriented programming class accessible has been an adventure for both the student and instructor and that cooperation was critical during the semester. We have been surprised as how limited the available tools are and have had to quickly improvise solutions for the problems as they arose. This paper is our attempt to convey the solutions we discovered. There are many talented individuals who have great potential in the field of Computer Science, but are discouraged as curricula become less, rather than more, accessible. Hopefully, others will be able to benefit from some of what we have learned and also contribute ideas and we can increase the participation of blind students in the field.

7 References

[1] "Annual Report 2012: Distribution of Eligible Students Based on the Federal Quota Census of January 3, 2011 (Fiscal Year 2012)," American Printing House for the Blind, 2012.

[2] J. P. Bigham, M. B. Aller, J. T. Brudvik, J. O. Leung, L. A. Yazzolino and R. E. Ladner, "Inspiring blind high school students to pursue computer science with instant messaging chatbots," in *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education*, New York, NY, 2008.

[3] A. Stefik, C. Hundhausen and D. Smith, "On the Design of an Educational Infrastructure for the Blind and Visually Impaired in Computer Science," in *Proceedings of the 42nd ACM technical symposium on Computer science education, 2011.*, New York, NY, 2011.

[4] J. Sanchez and F. Aguayo, "Listen what I do: Blind Learners Programming Through Audio," in *Memorias TISE 2004*, 2004.

[5] "Sodbeans Website," [Online]. Available: http://sodbeans.sourceforge.net/. [Accessed 2013].

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[7] "wxWidgets," [Online]. Available: http://www.wxwidgets.org/. [Accessed 2013].

[8] Visual Paradigm, Inc., "UML Modeling Tool with ERD and Code Generation," [Online]. Available: http://www.visual-paradigm.com/product/?favor=vpuml. [Accessed 2013].

[9] "Vim (text editor)," [Online]. Available: http://en.wikipedia.org/wiki/Vim_(text_editor). [Accessed 2013].

[10] "The Vinux Project," [Online]. Available: http://vinuxproject.org/. [Accessed 2013].

[11] "Orca Screen Reader," [Online]. Available: https://wiki.gnome.org/Orca. [Accessed 2013].

[12] "Tiger Elite200 Braille Printer," [Online]. Available: http://www.viewplus.com/products/braille-printers/elite-braille-printers/.

[13] "Accessibility | The Linux Foundation," [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/accessibility. [Accessed 2013].

[14] S. Yusuf, H. Kagdi and J. I. Maletic, "Assessing the Comprehension of UML Class Diagrams via Eye Tracking," in *15th IEEE International Conference on Program Comprehension, 2007. ICPC '07*, Banff, Alberta, BC, 2007.

[15] D. Schwab, "Non-Visual Diagram Navigation Using Microsoft's InkAnalysis Tool," Honor's Thesis, Winona State University, Department of Computer Science, 2006.

[16] D. L. Mansur, M. Blattner and K. Joy, "Sound-Graphs: A numerical data analysis," *Journal of Medical Systems*, vol. 9, pp. 163-174, 1985.

[17] S. Brewster, "Nonspeech auditory output," in *The human-computer interaction handbook*, Hillsdale, NJ, USA , L. Erlbaum Associates Inc., 2002, pp. 220-239.