The Sonified Omniscient Debugger: A Program Execution and Debugging Environment for Non-Sighted Programmers Built from the Ground up

Andreas Stefik*, Christopher Hundhausen*, and Robert Patterson†
Visualization and End User Programming Laboratory
**School of Electrical Engineering and Computer Science*
†*Department of Psychology*
Washington State University
Pullman, WA 99164−2752 USA
{astefik, hundhaus}@eecs.wsu.edu, rpatter@mail.wsu.edu

Abstract

Comprehending and debugging computer programs are inherently difficult tasks for sighted programmers. These tasks are even more difficult for non-sighted programmers, who must rely exclusively on audio-based representations of programs. The current state-of-the art approach to building program execution and debugging environments for non-sighted programmers is to retrofit existing visual environments with screen readers. Because of intrinsic differences in the ways humans process audio (serially) and visual information (in parallel), we argue that effective programming technologies for the non-sighted must instead be built from the ground up. To illustrate what such an environment might look like, we present the Sonified Omnicient Debugger (SOD), whose auditory cues were derived from empirical studies in which sighted proxies performed audio-based program comprehension tasks. In an experimental study involving sighted proxies, we compared SOD to (a) an identical environment with auditory cues from a state-of-the-art screen reader, and (b) an identical visual environment. Our results show that, for program comprehension and debugging tasks, SOD is significantly more effective than the environment with auditory cues from a state-of-the-art screen reader.

1. Introduction

Program comprehension and debugging are notoriously difficult tasks for sighted programmers. The difficulty of those tasks increases markedly for non-sighted programmers, who must depend exclusively upon audio-based representations of programs and their execution.

In contrast to the visual representations of programs that sighted programmers process in parallel (e.g., formatted source code, variable watch windows), audio-based representations must be processed serially. Given the inherent differences in how humans process audio and visual information, we find it surprising that the present state-of-the-art programming environment for the non-sighted consists of a visual environment (e.g., Microsoft® Visual Studio®) retrofitted with a screen reader (e.g., JAWS®). Indeed, programming environments for the blind that merely translate visual environments into audio would appear to be insensitive to the limits of human auditory processing and working memory. Therefore, they would appear to run the risk of overwhelming their users, diminishing their effectiveness as comprehension and debugging aids.

Is this actually the case? While gaining access to the blind programming community is challenging due to its broad geographical dispersion and relatively small numbers, our involvement with an international mailing list for blind programmers suggests that the blind programming community is generally unsatisfied with the current state-of-the-art. Indeed, the only mechanism the community has to improve usability is to use JAWS® scripts to customize the behavior of the JAWS® screen reader. However, these community-created scripts do not have the ability to customize the delivery of the complex information processed by compilers and debuggers; instead, they are generally used only to improve simpler issues like navigation in the environment.

In order to remedy deficiencies in the state-of-the-art program execution and debugging environments for the blind, we believe that one needs to fundamentally rethink the current approach to the problem. Rather than attempting to make visual environments accessible to the non-sighted by retrofitting them with screen readers, we believe that a superior design strategy is to build such environments for the blind from the ground up. This means designing for the non-sighted from the start through a user-centered design process [1] driven by empirical studies of the non-sighted—or if non-sighted programmers are unavailable, then at least of sighted proxies, who have been shown to exhibit performance that reasonably approximates that of the non-sighted [2].

We have been using such an approach to explore the design of auditory cues to support program comprehension and debugging tasks. Our exploration has been driven by two main research questions:

RQ1: *What auditory cues best facilitate the comprehension of static program structure?*

RQ2: *What auditory cues best facilitate the comprehension of run-time program state and behavior?*

In this paper, we address these questions by presenting and experimentally evaluating a novel program execution and debugging environment called the Sonified Omniscient Debugger (SOD). The auditory cues supported by our environment were designed iteratively through empirical studies involving sighted proxies. To evaluate the effectiveness of the SOD environment, we experimentally compared it against (a) a visual environment (used to establish an upper bound on human performance) and (b) a current state-of-the art environment for the blind: Microsoft® Visual Studio® 2005 coupled with the JAWS® 9 screen reader. In our study, the SOD environment promoted significantly more efficient task performance than did Visual Studio coupled with JAWS. Moreover, whereas a significant difference was found between the visual environment and Visual Studio + JAWS, no such difference was found between the visual environment and SOD. These results indicate that SOD constitutes a significant improvement over the current state-of-the-art.

The remainder of this paper is organized as follows. After reviewing related work in Section 2, we describe the design of the SOD environment in Section 3. Then, in Section 4, we present the design and results of our experimental evaluation. Finally, in Section 5 we summarize our contributions, and outline directions for future research.

2. Related Work

Auditory display techniques use audio to represent either data or information about data [3]. Gaver's [4] pioneering work on SonicFinder, a system that used audio to provide a richer and more informative desktop interface, introduced the idea of *perceptual mappings* between data and sound, and defined three different types: *symbolic*, *metaphorical*, and *iconic*.

Auditory displays have been developed to assist the blind and visually impaired in a variety of tasks, including navigating web pages [5], reading tables [6], comprehending mathematics [7], browsing molecular structures [8], and improving short term memory [9].

While computer programming, debugging, and comprehension tasks have been widely studied over the past three decades, relatively little research has considered the use of assistive auditory technologies for these tasks. Begel and Graham [10] created Spoken Java, a tool designed to allow users to speak computer code instead of typing it. This work can be characterized as the reverse of the work presented here. Spoken Java provides a means of inputting computer programs via speech, whereas our work provides a means of outputting computer programs as speech.

Boardman [11] created the language LISTEN to explore alternative mappings between source code and sound. In contrast to our work, LISTEN focused on facilitating code-to-audio mappings, not on the design of the auditory cues.

Brown and Hershberger [12] augmented algorithm animations in their Zeus system with "algorithm auralizations." Their musical auditory displays mapped higher-pitched tones to larger magnitude data in the algorithms being auralized. Brown and Hershberger claimed that their auditory displays assisted in the comprehension of the algorithm animations by communicating patterns, conveying additional information not depicted in the animations, and signaling exceptional conditions; however, they did not carry out any empirical evaluations to test these claims.

In a similar vein, Vickers [13] built a program auralization system called CAITLIN, which used musical auditory cues to represent the execution of computer programs written in Pascal. In experimental studies, Vickers failed to show that participants could find more bugs with musical cues, although he claimed to have found that the effectiveness of the musical cues increased with the cyclomatic complexity of the source code. Subsequent work has shown that musical auditory cues are, in fact, difficult to learn [14]. This line of work differs from ours in that we explore the use of speech-based cues as opposed to musical cues.

In a line of research perhaps most related to the work presented here, Smith *et al.* [15] developed a speech-based hierarchical structure analysis tool, *JavaSpeak*, specifically for non-sighted computer programmers. This tool was informally tested in empirical studies by having sighted proxies and two blind students navigate complex hierarchical structures. Whereas Smith *et al.* focused specifically on auralizing hierarchical structures, we have built an entire program debugging environment for the non-sighted from the ground up.

3. The Sonified Omniscient Debugger

Traditional assistive screen readers for non-sighted computer users are driven by information made available through accessibility APIs. While screen readers grant blind or visually impaired users basic access to visual interfaces, such "retrofitted" interfaces are clearly not optimized for the usability and efficiency of non-sighted users, because they fail to acknowledge inherent differences in the ways humans process visual and auditory information. Rather than retrofitting visual interfaces, we believe a better design strategy is to build program execution and debugging environments for the blind from "the ground up." To illustrate how one might do that, this section presents the Sonified Omniscient Debugger (SOD), a new program execution and debugging environment for non-sighted programmers.

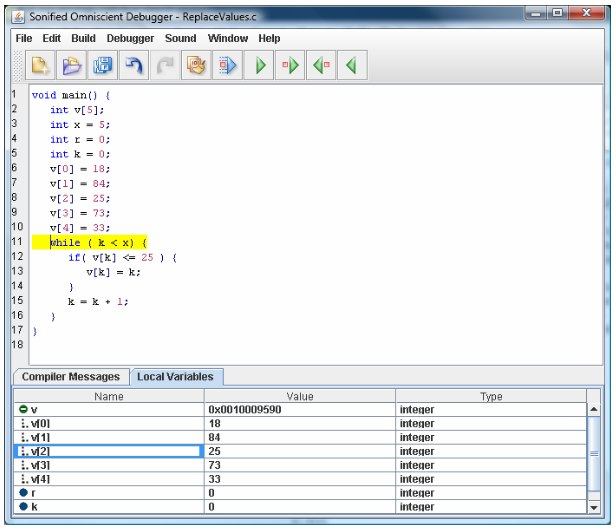
As the name implies, SOD is both a sonified debugger and an omniscient debugger. SOD is a sonified debugger because it allows for any possible run-time or static attribute of a computer program to be presented to the user via auditory cues. SOD is an omniscient debugger [16] because, as SOD executes a program, it stores a record of computation and all changes to the state of the program. As a result, the environment is able to execute programs in reverse, and to provide users with information on program execution history.

3.1 Sample Session with SOD

In order to provide a feel for how the SOD environment works, we now walk through a sample session in which we use the SOD environment to complete a portion of one of the tasks performed by participants in our experimental evaluation (see Section 4). The task is to answer a comprehension question about the "Replace Values" algorithm, which replaces values in an array that are less than 25 with zero. For comparison purposes, we describe in this example both SOD's auditory cues, and the state-of-the-art auditory cues that are generated by the JAWS® 9 screen reader coupled with Microsoft® Visual Studio® 2005.

Figure 1 presents a visual snapshot of the SOD environment. The C source code for the same buggy "Replace Values" algorithm used in our experiment is loaded into the environment. In this example, we consider how a user might answer the comprehension question, "How many items have been replaced after the third iteration of the loop?" The user could answer this question using three different methods. Illustrating these three methods will expose us to SOD's complete repertoire of auditory cues.

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



```
1  void main() {
2     int v[5];
3     int x = 5;
4     int r = 0;
5     int k = 0;
6     v[0] = 18;
7     v[1] = 84;
8     v[2] = 25;
9     v[3] = 73;
10    v[4] = 33;
11    while ( k < x) {
12       if( v[k] <= 25 ) {
13          v[k] = k;
14       }
15       k = k + 1;
16    }
17 }
18
```

Name	Value	Type
v	0x0010009590	integer
v[0]	18	integer
v[1]	84	integer
v[2]	25	integer
v[3]	73	integer
v[4]	33	integer
r	0	integer
k	0	integer

Figure 1. Snapshot of SOD interface while performing "replace values" comprehension and debugging task

The second method the user could employ would be (a) to execute the loop three times by repeatedly pressing the F9 key to forward execute each line of code, and then (b) to count the number of times the assignment statement that replaces array values (line 13 in Figure 1) is executed. To illustrate how this would be done, let us assume that the user is positioned at the `while` statement highlighted in Figure 1. From there, the user would need to press F9 four times in order to navigate through one complete loop iteration.

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

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

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

What does *"graphic 53"* mean in the preceding auditory cue? It turns out that when Visual Studio + JAWS reads the line that is about to be executed, it literally translates the execution arrow as *"graphic 53."* In contrast, when a user navigates to the line that is about to be executed in SOD, the user hears *"cursor at the execution line"* followed by a semantic depiction of the line itself (e.g., *"if true"*).

A third method for answering the question "How many items have been replaced after the third iteration of the loop?" is identical to the second method just described, except that the user would use the Locals Window (bottom of Figure 1) to inspect the contents of the array after completing three loop iterations, rather than keeping track of the number of times that the set statement on line 13 executes.

In both the SOD and Visual Studio + JAWS interfaces, the user can navigate to the Locals Window by pressing ALT + F3, at which point both SOD and Visual Studio + JAWS say *"locals."* Once in the Locals Window, users of both interfaces can navigate from variable to variable using the up and down arrows. When a variable is visited, the variable name, value, and type are read in sequence. For example, suppose the user hits the down arrow to visit variable `r` (second from the bottom in Locals Window of Figure 1). In both interfaces, the user would hear *"r zero integer."*

The auditory cues for array variables differ slightly between SOD and Visual Studio + JAWS. To illustrate, suppose that the user navigates to the array variable `v` (top-most variable in Locals Window of Figure 1). In both interfaces, the user would hear the following: *"v zero x zero zero one zero zero zero nine five nine zero integer"* (the long sequence in the middle is the memory address of the array). In order to see the individual array values of `v`, the user would need to open the array by hitting the right arrow key. When the user does this in the SOD interface, the user hears *"opening array v"*. In contrast, hitting the right arrow key to open an array in Visual Studio + JAWS does not generate an auditory cue. Once the array is opened, the user can navigate to individual array values with the up and down arrows. The auditory cues produced for array values are identical to those produced for regular variables, with one notable exception: SOD says an array variable using "sub" language (e.g., *"v sub zero"*), whereas Visual Studio + JAWS says an array variable by speaking it literally (e.g., "*v left bracket zero right bracket"*).

3.2 Building SOD from the Ground up

The example just presented highlights several notable differences between the auditory cues produced by SOD and the state-of-the-art auditory cues produced by Visual Studio + JAWS. These differences are no accident. Rather, they are the result of a user-centered design process [1] that has focused on designing and testing highly comprehensible auditory cues independently of a visual execution environment.

This process has included numerous formative and summative empirical studies involving sighted proxies. In these studies, a fundamental challenge has been to measure the comprehension of auditory cues. To address this challenge, we have developed *artifact encoding*, a free-form writing and coding paradigm that allows us to gauge how well a participant interprets a given auditory cue within the context of a computer program.

While the details of our artifact encoding paradigm are beyond the scope of this paper (see [17]), the basic idea is to give participants a sequence of auditory cues that represent a computer program, and then to ask them to write down what they thought the cues meant. Using techniques taken from the bioinformatics literature on DNA sequence matching [18], we can automatically grade and statistically analyze participants' answers to provide a detailed assessment of their comprehension.

Artifact encoding is the basis of our "ground up" approach to building SOD's auditory cues. We have been iteratively refining our *behavioral runtime cues* (the ones generated when F9 is pressed) for over a year, and have recently completed a large scale, formal evaluation of those cues [17]. However, SOD's *editing cues* (the ones generated when the arrow keys are pressed) are in the early stages of design, and have not yet been empirically validated through artifact encoding studies.

4. Experimental Evaluation

In order to evaluate the effectiveness of the SOD environment, we conducted an experimental study with the following hypothesis:

H1: *In program comprehension and debugging tasks, SOD's auditory cues will promote significantly faster and more accurate performance than the auditory cues generated by Visual Studio + JAWS; however, a visual environment will promote significantly more efficient performance than both SOD and Visual Studio + JAWS.*

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

1. *SOD*: An audio-only version of the environment presented in the previous section embedded with the SOD auditory cues;
2. *JAWS*: An audio-only version of the environment presented in the previous section embedded with the auditory cues produced by JAWS® 9 in Visual Studio® 2005 (the current state-of-the-art); and
3. *Visual*: The visual environment presented in the previous section with no auditory cues. This can be considered the "gold standard" that establishes an upper bound on human performance.

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

4.1 Participants

We recruited 19 students (13 male, 6 female; mean age 22.9) out of the Spring, 2008 offering of CptS 443/580, the undergraduate/graduate human computer interaction course at Washington State University. Participants were either juniors, seniors, or graduate students, and reported a mean of 5.47 years of prior programming experience. One participant, despite being told otherwise, thought aloud, which is known to be a confound for experiments involving audio interfaces [19]. Another participant failed to complete the tasks, and was a significant outlier. These two participants were removed from the statistical analysis.

4.2 Materials and Tasks

All participants worked on a computer running the Windows XP operating system. While the machine was equipped with a mouse and keyboard, participants were allowed to use only the keyboard. In the Visual condition, a 19" LCD color display was set to a resolution of 1024 × 768. In the audio conditions, the Microsoft Mary voice was used in the text-to-speech speech engine.

Prior to working on the tasks in each condition, participants completed an informationally-equivalent training task that introduced them to the environment version they would be using by having them navigate the code structure, run the debugger, and use the local variable window, either with visual or auditory feedback.

Participants worked with three different program execution and debugging environments defined by condition: SOD, JAWS, and Visual. All three of these environments were identical with respect to the input commands they supported. They differed only with respect to the output they used to communicate with the user: the SOD environment used our own experimental cues; the JAWS environment used the cues generated by JAWS® 9 in Visual Studio® 2005; and the Visual environment presented the visual interface depicted in Figure 1, but had no auditory cues.

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

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

4.3 Procedure

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

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

4.4 Results

Before analyzing our data, we first verified that there were no order or task effects. An analysis of variance (ANOVA) found no significant differences in (a) time for the total order, *F*(7,43) = .927, *p* = .495; (b) performance in the three tasks (Find Max, Count, Replace) *F*(2,48) = .345, *p* = .710; and performance with respect to task sequence, *F*(2,48) = .026, *p* = .974. We conducted the same analysis on the comprehension accuracy and debugging accuracy metrics, but similarly found non-significant order and task effects. We thus concluded that our tasks were reasonably isomorphic and that our Latin square successfully removed order effects from our design. In addition, using normal probability plots, we confirmed that our accuracy and time-on-task data were normally distributed.

Figure 2 and Figure 3 plot time on task, comprehension accuracy, and debugging accuracy by condition. Not surprisingly, these figures indicate that participants in the Visual condition promoted better performance than the other two conditions with respect to all three dependent measures. Moreover, the SOD condition appears to have promoted better performance than the JAWS condition in all three measures.

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

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

With respect to debugging accuracy, the overall ANOVA model was non-significant, *F*(2,48) = 1.606, *p* = .211 (partial eta-squared = .063).

Finally, we wanted to see if participants' subjective opinions about each environment accorded with their objective performances with each environment. Figure 4 plots participants' responses to five exit questionnaire questions designed to elicit participants' subjective opinions on each environment. All responses were on a Likert scale ranging from 1 (strongly disagree with statement) to 7 (strongly agree with statement).

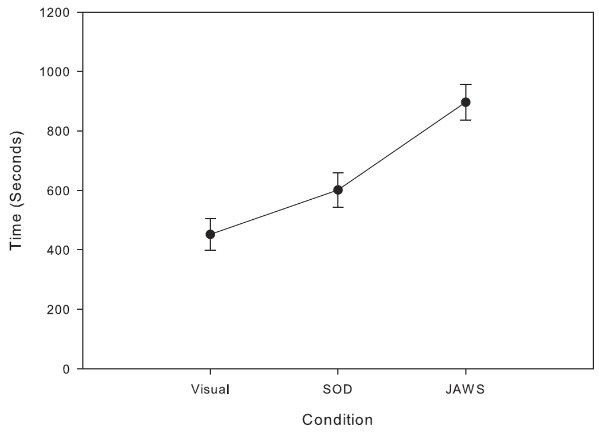


Figure 2. Total time on task by condition

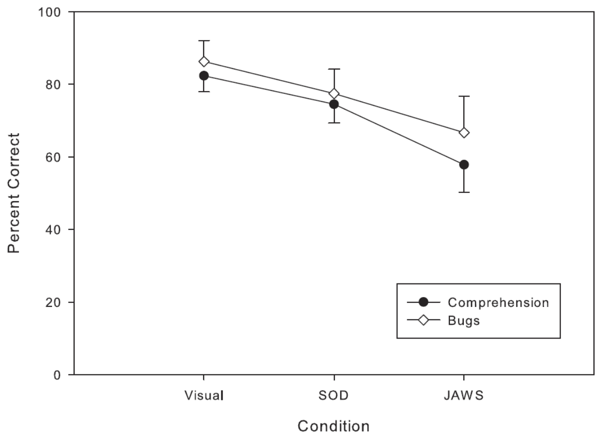


Figure 3. Comprehension and debugging performance by condition

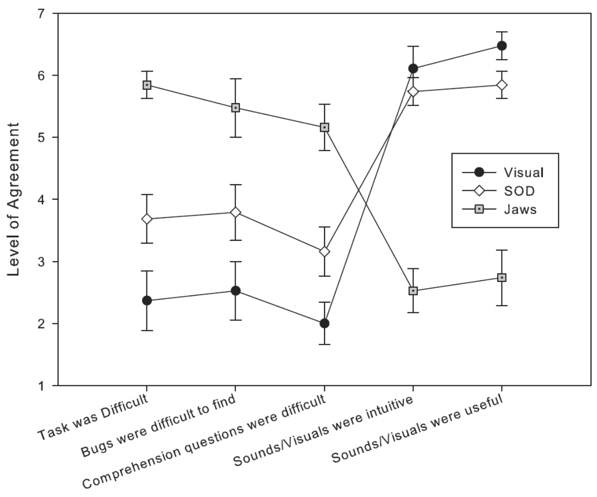


Figure 4. Exit questionnaire results by condition

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

4.5 Discussion

Our results provide strong empirical support for our hypothesis that the SOD auditory cues promote significantly faster performance than the JAWS auditory cues. Indeed, the JAWS group took 4.9 min longer to complete the comprehension and debugging tasks than the SOD group—a difference that is both statistically and practically significant.

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

With respect to comprehension and debugging task performance, we found only one statistically significant difference between the Visual and JAWS groups. Especially in the debugging task, we suspect that the lack of differences had to do with a ceiling effect: There were only two bugs to find within algorithms totaling fewer than 20 lines each, and most participants succeeded in finding the bugs.

Why was it the case that our main hypothesis—that the SOD auditory cues hold a significant advantage over the JAWS auditory cues—was substantiated? We can offer two primary theoretical explanations *semantic priming* and *temporal masking*.

Semantic priming [20] is a psychological theory that suggests that a participant's response to one stimulus (e.g., an auditory cue) can alter the speed at which the participant can mentally retrieve, through a process called *spreading activation*, another stimulus. Why would semantic priming afford the SOD auditory cues an advantage? Consider the auditory cues generated when the participants execute a line of code (by pressing the F9 key). In the JAWS environment, the auditory cue is *"F9*," which has no meaning in the context of the executing program. In contrast, in the SOD environment, the auditory cue provides information relevant to context, such as "*if true*." This information may prime the participant's mental model of both the executing program's behavior, as well as the participant's temporal location within that program.

In contrast, temporal masking [21] is the theory that a sound stimulus can have an effect on the audibility of another stimulus either before or after the original**.** We suspect this phenomenon relates to the auditory cues generated when the user is navigating the code with the arrow keys. When auralizing a line of code, the JAWS environment reads every character literally, including brackets, braces, semicolons, and parentheses. It is possible that this large influx of superfluous details masks the information most relevant to the user, such as whether the line has a loop or conditional in it.

After reviewing our recordings of the experimental sessions, we find this explanation based on temporal masking to be especially plausible. In the JAWS sessions, participants would often have to listen to the auditory cue for each line multiple times. In contrast, in sessions with the SOD environment, which removed these extra characters and made them available only by pressing the left and right arrows, we noticed that participants rarely had to listen to the same line multiple times.

5. Summary and Future Work

While a large body of research has explored the design of visual representations to aid in computer programming tasks, relatively little research has explored the design of auditory representations to assist non-sighted programmers. Rather than retrofitting existing visual environments with screen readers, we have argued that a superior design strategy is to build auditory representations "from the ground up"—that is, through a user-centered design process that iteratively tests auditory cues independently of existing visual environments. We have presented SOD, an auditory program execution and debugging environment that evolved out of such a process, and we have furnished empirical evidence that it promotes significantly more efficient human performance in comprehension and debugging tasks than JAWS® 9 coupled with Microsoft® Visual Studio 2005—the current state-of-the-art program execution and debugging environment for the non-sighted.

In future work, we would like to explore two different directions. First, at present, only SOD's *behavioral runtime cues*, which provide information about what an executing computer program just did, have undergone substantial iterative design and testing. In future research, we would like to subject a broader range of SOD's auditory cues to iterative design and empirical testing—most importantly, cues for source editing and browsing the Locals Window.

Second, a clear limitation of the present SOD environment is that it supports only single-procedure algorithms. In future work, we would like to expand the SOD environment so that it supports large numbers of source files containing multiple, complex methods. Only then will we be able to explore program navigation techniques and auditory cues that truly support the authentic programming-in-the-large practices in which the blind programming community engages.

6. Acknowledgments

We are grateful to the members of the Blind Programmers Group (http://www.blindprogramming.com) for providing valuable feedback on this research, and to students in the spring, 2007 offering of CptS 443/580 ("Human-Computer Interaction") at Washington State University for participating in our experimental study.

7. References

[1] D. Norman and S. Draper, *User-centered system design*. Lawrence Erlbaum Assoc., Mahwah, NJ, 1986.

[2] L. H. D. Poll, "Visualizing Graphical User Interfaces for Blind Users," Ph.D. Thesis, Technische Universiteit Eindhoven, 1996.

[3] B. N. Walker and G. Kramer, "Mappings and metaphors in auditory displays: An experimental assessment," *ACM Trans. Appl. Percept. 2* (4), 2005, pp. 407-412.

[4] W. W. Gaver, "The Sonic Finder: an interface that uses auditory icons," *Human-Computer Interaction 4*, 1989, pp. 67-94.

[5] P. Parente, "Audio enriched links: web page previews for blind users," *SIGACCESS Access. Comput.* (77-78), 2004, pp. 2-8.

[6] Y. Yesilada, R. Stevens, C. Goble, and S. Hussein, "Rendering tables in audio: the interaction of structure and reading styles," *SIGACCESS Access. Comput.* (77-78), 2004, pp. 16-23.

[7] R. Stevens, "Principles for the Design of Auditory Interfaces to Present Complex Information to Blind People," Ph.D. Thesis, The University of York, 1996.

[8] A. Brown, S. Pettifer, and R. Stevens, "Evaluation of a non-visual molecule browser," in *Proc. ACM SIGACCESS Conference on Computers and Accessibility*. ACM Press, New York, 2004, pp. 40-47.

[9] J. Sanchez and H. Flores, "Memory enhancement through audio," *SIGACCESS* (77-78), 2004, pp. 24-31.

[10] A. Begel and S. L. Graham, "An Assessment of a Speech-Based Programming Environment," in *Proc. 2006 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE CS Press, Los Alamitos, 2006, pp. 116-120.

[11] D. B. Boardman, G. Greene, V. Khandelwal, and A. P. Mathur, "LISTEN: A tool to investigate the use of sound for the analysis of program behavior," in *Proceedings Nineteenth Annual International Computer Software and Applications Conference*. IEEE CS Press, Los Alamitos, CA, 1995, pp. 184-189.

[12] M. H. Brown and J. Hershberger, "Color and sound in algorithm animation," *IEEE Computer 25* (12), 1992, pp. 52-63.

[13] P. Vickers and J. L. Alty, "Musical program auralization: Empirical studies," *ACM Trans. Appl. Percept. 2* (4), 2005, pp. 477-489.

[14] D. K. Palladino and B. N. Walker, "Learning rates for auditory menus enhanced with spearcons versus earcons," in *Proceedings of the 13th International Conference on Auditory Display*. Montreal, CA, 2007, pp. 274-279.

[15] A. C. Smith, J. S. Cook, J. M. Francioni, A. Hossain, M. Anwar, and M. F. Rahman, "Nonvisual tool for navigating hierarchical structures," in *Proc. ACM SIGACCESS Conference on Computers and Accessibility*. ACM Press, New York, 2004, pp. 133-139.

[16] B. Lewis and M. Ducassé, "Using events to debug Java programs backwards in time," *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 96-97.

[17] A. Stefik, "On the design of program execution environments for non-sighted computer programmers," Ph.D. Thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, in preparation.

[18] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology 48*, 1970, pp. 443-453.

[19] S. Tsujimura and Y. Yamada, "A study on the degree of disturbance by meaningful and meaningless noise under the brain task," in *19th International Congress on Acoustics*, Madrid, Spain, 2007.

[20] P. Whitney, *The Psychology of Language*. Houghton Mifflin Company, Boston, 1998.

[21] C. F. Zhang and C. Formby, "Effects of cueing in auditory temporal masking time," *Journal of Speech, Language, and Hearing Research 50* (3), 2007, pp. 564-575.