A Voice-activated Syntax-directed Editor for Manually Disabled Programmers

Thomas J. Hubbell
TAK Imaging
Fairhope, AL 36532
1-251-990-4408
thomas.hubbell@takimaging.com

David D. Langan
Sch. of Computer & Info. Sciences
Univ. of South Alabama
Mobile, AL 36688
1-251-460-6390
dlangan@usouthal.edu

Thomas F. Hain
Sch. of Computer & Info. Sciences
Univ. of South Alabama
Mobile, AL 36688
1-251-460-6390
thain@usouthal.edu

ABSTRACT
This paper discusses a research project targeted at the design and implementation of an interface intended to allow manually disabled people to more easily perform the task of programming. It proposes a Speech User Interface (SUI) targeted for this task. Voice was selected as the means of input as an alternative to the keyboard and mouse. Traditional programming IDEs tend to be character and line oriented. It is argued that this orientation is not conducive to voice input, and so a syntax-directed programming interface is proposed. To test the viability of this combination of voice with a syntax-directed approach, an editor named VASDE (Voice-Activated Syntax-Directed Editor) was implemented using ECLIPSE as the underlying platform for development. This paper describes the syntax-directed interface, VASDE, and some of the lessons learned from initial usability studies.

Categories and Subject Descriptors
H.5.2 [**User Interfaces**]: voice IO, graphical user interfaces, interaction styles, input devices and strategies.

General Terms
Design, Experimentation, Human Factors, Languages.

Keywords
Speech user interface, syntax directed, programming by voice, IDE.

1. INTRODUCTION

1.1 Motivation
With the point-and-click interface of a mouse-driven GUI supplemented with a keyboard, it might appear that there would be no remaining barriers to accessing modern computer applications. However, for manually disabled users the keyboard/mouse interface is largely or totally inaccessible. For users with such physical disabilities as carpal tunnel syndrome, the hand and arm movements required by such interfaces may be difficult, painful, or impossible.

Programmers are among those computer users often affected by such limiting disabilities. Due to their prolonged and heavy computer usage, they are at risk for a family of ailments known as repetitive strain injuries (RSI). RSI is defined as "a soft-tissue disorder that results from the repetitive use of some part of the body" [8]. Repetitive Strain Injuries, such as carpal tunnel syndrome, are serious conditions that can lead to numbness, pain, and, in extreme cases, paralysis [8]. Although recent research has shown that carpal tunnel syndrome is not directly caused by prolonged computer usage, it is known that extended computer use can exacerbate the symptoms of existing carpal tunnel syndrome [16]. Furthermore, repeated computer use is likely to cause other ailments, such as tendonitis [16], which can lead to a painful computing experience.

1.2 Problems with Programming by Dictation
One viable input method for users with hand/arm disabilities is speech. Speech-recognition/dictation software packages are available that can supplement such applications as email, or word processors. However, most of these accessibility-enabled applications take a pre-existing interface, and simply layer voice support over it. This approach ignores the possibility that an interface designed from the outset for voice input might be constructed to look and feel more natural, and be more efficient.

The speech interface requirements for a programmer are different from those of the user of a typical application. A dictation interface to a word processing application makes use of a large set of known words in the user's spoken natural language, as well as standard punctuation required for sentence structure. In contrast, programming editors typically involve a limited number of programming language keywords (e.g., "if", "else", "while", etc), a variety of punctuation marks, and a relatively wide variety of user-defined identifiers typically not found in any natural language (e.g., "myPointer", "getCount"). Thus, for new identifiers it is necessary to resort to a "spelling interface" of the voice recognition system.

Clearly, in order to make speech recognition a viable input method for programmers, some approach other than pure dictation must be used.

ASSET'06, October 22–25, 2006, Portland, Oregon, USA.

1.3 Research Overview

The objective of this research was to create a programming editor with the following characteristics:

1. Accessible to manually disabled programmers.
2. Intuitive to use, after a minimal initial training.
3. Efficient, in the sense of programmers being able to generate, and edit, code quickly.
4. Effective, in the sense of reducing the probability of programmer-generated syntax errors.

The goal of this research was to evaluate whether a speech interface coupled with a syntax-directed editing approach would provide a "natural" match in the domain of programming editors. It was recognized that the well-defined grammar of programming languages provides a limited number of input possibilities at any given time. These possibilities can be in some way enumerated by the editor, and selected vocally by the programmer. In many instances the possibilities are singular, in which case the editor can insert that text automatically. The combination of how the screen presents the current choices to the programmer along with a paradigm of how the speech is to be used forms a Speech User Interface (SUI). Thus while investigating the use of speech in general, we investigate the issue of refining the SUI to be as effective as possible. To provide a usability testing environment for this SUI concept, such a voice-activated syntax-directed editor (named VASDE) was designed and implemented.

While objective 1 was paramount, it was felt conceivable that the interface might be useful to the programmer population at large, and that a multi-modal input interface—as long as it did not compromise accessibility for manually disabled programmers—may be possible. Objective 2 was ensured by staying as close to standard interface conventions as possible, only deviating where some advantage could be gained for the other objectives. Objective 3 would be supported by making every attempt at reducing the vocal bandwidth (defined as the average number of spoken characters per program character). Both objectives 3 and 4 are supported by the syntax-directed approach, since many lexical and syntactical elements can be automatically inserted, and because selection (of syntax-directed elements) is less error prone, and requires smaller vocal bandwidth than, for example, keyboard entry.

Section 2 discusses related research. Section 3 describes the VASDE editor that was implemented based on the proposed interface. Section 4 discusses the results and conclusions of the VASDE usability study.

2. RELATED RESEARCH

Previous research related to the design of a voice-activated programming interface can be categorized into four main groups: (a) UI research examining effective ways of utilizing speech recognition, (b) research into syntax-directed editing approaches, (c) prior research directly related to programming-by-voice (PBV) issues, and (d) research regarding the practical implementation of speech-recognition technology. Some results from each of these are briefly discussed below.

2.1 SUI Research

In the general area of SUI research, Oviatt [14] presents research focused on "constrained" (i.e., guided) versus "unconstrained" (i.e., unguided) speech interfaces. Oviatt found that "a more structured interface reduced the number of words, length of utterances, and amount of information integrated into a single utterance." Furthermore, they found that users preferred the constrained interfaces to the unconstrained ones by a factor of two to one. These findings have a direct bearing on the interface design for a programming editor, as a highly structured interface can reduce the computational requirements of the speech-recognition engine and can result in a generally more pleasant user experience.

2.2 Syntax-directed Editing

In the area of syntax-directed editing environments, Horwitz and Teitelbaum [10] present a design for a language-independent model of editing involving the representation of programs as attributed abstract-syntax trees with an associated relational database. Arefi [1] further expands upon the basic tree view by specifying languages as directed, labeled graphs, which allowed for quick updates to the program structure. Reiss [15], in his PECAN system, explores a variety of different views of a program that could be achieved with a syntax-directed editing paradigm. Steindl [17] also explores the flexibility of the syntax-directed approach in proposing data dependency views and links between procedure calls and definitions. Biddle et al. [2] proposes an interesting syntax-directed view in the "Dependency Visitor", which provides a tree view of a program organized by scope. In the dependency tree, subtrees are major program elements such as object variables, object methods, and method bodies. The leaves of the tree are variable definitions or type indicators (for methods). Biddle et al. further proposes a selection mechanism that relies on program units (such as the name of an object or the name of a method), rather than arbitrary characters.

After the VASDE application had been nearly completed and this research effort was coming to a close, a parallel effort was discovered that exhibited many similarities to VASDE. The "Happy Hands Java Speech Editor" [9] is a completely independent commercial effort that co-evolved as VASDE was developed. It was developed to solve the same problem as VASDE – to create an effective programming-by-voice environment for Java developers. While VASDE and Happy Hands both share a similar syntax-directed, template-based SUI approach, these approaches are realized in very different ways in the respective UIs. In comparison to Happy Hands, VASDE's approach is quite unique in the areas of tree-based navigation and selection as well as the ability to create new names by voice.

2.3 Programming by Voice

In the area of programming-by-voice research, Johannsson [12] presents the idea of using templates to limit the syntactical elements entered directly by the programmer. The VoiceCode project [18][6][3][4] explores numerous approaches to programming by voice in their attempt to modify an existing programming editor for use with voice.

2.4 Speech Recognition Technology

In research regarding the practical implementation of speech-recognition technology, Sun Microsystems provides some general guidelines for the implementation of speech technology [11] in any application. These guidelines address key areas including performance and the choice of a proper spoken command set.

3. VASDE (Voice-Activated Syntax-Directed Editor)

3.1 Overview

VASDE is a voice-activated syntax-directed programming editor. Because this application was intended as a research tool rather than a commercial IDE, a few simplifying design decisions were made:

- The editor is bound specifically to the Java programming language.
- Although Java is the target language, some features/constructs were not included if their omission represented no loss of generality.
- All high-level tasks within the editor were designed to be accessible by means of a speech interface. However, the current implementation of VASDE does not address the task of expression editing. We are currently exploring this aspect—also based on a syntax-directed approach—and are addressing the lower-level issues that arise with expressions.

The Eclipse open source project is used as the underlying platform for the VASDE implementation [5]. Eclipse was originally conceived as a type of extensible IDE environment, although it has since expanded its purview into numerous other areas. Thus, it provides a set of libraries, frameworks, and applications.

The Eclipse project provides a full-featured Java programming IDE known as the JDT. JDT contains a number of useful libraries for dealing with Java programs, including abstract syntax tree APIs, compiler abstractions, and the definition of a "Java project"—a collection of the source and resource files which comprise a Java application.

VASDE borrows heavily from this project in its infrastructure, and, in fact, is designed as an application plug-in to the Eclipse environment itself. The specific Eclipse items used within VASDE are as follows:

- Basic application framework via the Eclipse Rich Client Platform (RCP)
- User-Interface Widgets via the Standard Widget Toolkit (SWT) and JFace libraries
- JDK infrastructure including:
  - Java Project Management framework
  - Java compilation, compilation error reporting, and error correction suggestions
  - Java Abstract Syntax Tree and support libraries

Another core component of VASDE is obtained from an existing technology. The Java Speech API (JSAPI) [11] provides the interface between VASDE and the underlying speech recognition engine by providing a means for recognizing speech, and triggering required actions. It also provides a means to specify a "grammar" of speech commands that would be matched by the speech engine. The specific implementation of this API is provided by a company named CloudGarden, whose JSAPI implementation is speech engine-neutral. Finally, because of its widespread availability, the IBM ViaVoice engine was selected for use with VASDE.

3.2 VASDE Details

The main infrastructure of VASDE was created by combining relevant portions of Eclipse and the JSAPI together and augmenting the resulting constructs with program (the program being edited) and application (VASDE itself) state management objects. These infrastructure components provided the foundation for the presentation of the user interface – the most important aspect of VASDE for this research effort.

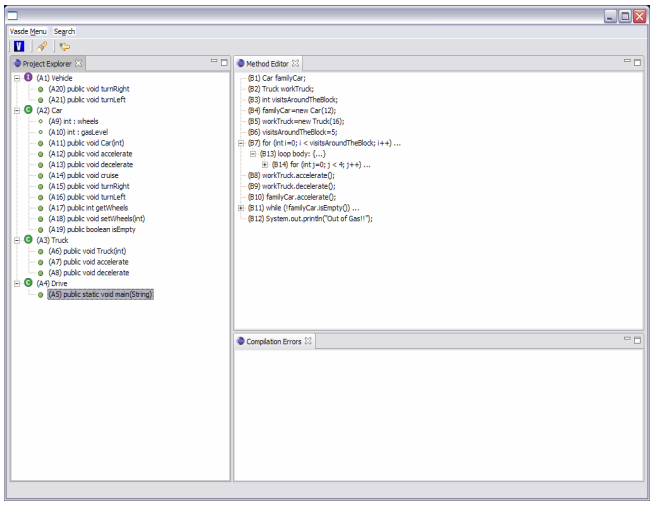

Figure 1 VASDE Main Application Interface.

There are four components to the user interface (seen in Figure 1):

1. The Application Framework provided by Eclipse
2. The Project Explorer view
3. The Method Editor view
4. The Compilation Errors view (a read-only display).

The Application Framework interface for VASDE refers to the main application shell, consisting of the main window, the menu bar, and the tool bar. The Application Framework actually hosts the other UI components seen in Figure 1. In VASDE, each of the hosted components is considered to be an Eclipse "view". Although the views combine with the framework to form a seamless interface, each view is a separate component. Therefore, each view has a separate user and speech interface. A small set of voice-enabled elements have been added to Application Framework itself, allowing the following actions:

- Open the **Project** dialog, which allows users to create a new project or open an existing project.
- Open a dialog which allows editing of the build path that the compiler uses for the currently loaded project.
- Exit the program.

The Project Explorer view (the leftmost component pictured in Figure 1) and the Method Editor view (the top-right component pictured in Figure 1) are both tree-viewer interfaces that share a nearly identical speech interaction paradigm. Both views highlight the syntax-directed nature of VASDE. Elements within these tree views correspond to meaningful syntactic elements in the program. Therefore, editing, viewing, and navigation of a program takes place by means of some syntactic element of the Java program—not a syntactically meaningless unit such as a line of

text. While the organization and selection of syntactic units takes place directly within these tree views, editing an individual syntactic unit is accomplished by means of a series of dialogs. These template-based dialogs generate the fixed elements (parentheses, braces, colons, keywords, etc.) of the syntactic unit, while requiring the programmer to enter only the elements that are specific to the current unit.

The Project Explorer enables exploration of top-level items within the currently open project, as selected by the Open Project dialog. Top-level items for a VASDE project include classes, interfaces, class fields, class/interface method signatures, constructor signatures, imports, and inner classes/interfaces. From this view, each of these items can be created and edited by invoking a specific dialog for each construct. Additionally, items can be deleted from within this view. Finally, fields, methods, imports, and inner classes/interfaces can be cut, copied, and pasted from one class/interface to another.

While the Project Explorer handles the organization of the program from the class level down to the method level, editing and navigation within individual method and constructor bodies is accomplished by means of the Method Editor. Here, the meaningful syntactic units are, for example, assignment statements, variable declarations, for loops, if statements, and method invocation statements. These items correspond to nodes that are included in the Eclipse definition of a Java abstract syntax tree.

The speech-enabled command sets of these two views have two general forms:

1. *command + label*
2. *command + labelX "**through"** labelY*

The command element is any of the defined actions, while the label is a short, unique identifier that is dynamically assigned. The first form is used to apply a single command to a single labeled node, while the second form is used to apply a single command to a sequential group of labeled nodes. Most of the commands are applicable to any node types, though some types recognize additional context-sensitive commands. The set of commands includes the following (in all but a few cases, both general forms are applicable):

- *"select"*
- *"expand"*
- *"contract"*
- *"add" **node type*—**a**pplies only to Project Explorer
- *"insert" **node type** ("before" | "after")*—applies only to Method Editor
- *"delete"*
- *"copy"*
- *"cut"*
- *"paste"*—applies only to Project Explorer
- *"paste into"*—applies only to Project Explorer
- *"paste" (before | after)"*—applies only to Method Editor
- *"edit"*
- *"edit (method | constructor) signature"*—applies only to Project Explorer; used to invoke the Method Editor

Particular commands are only applicable to a specific view since the order of nodes does not matter in the Project Explorer view (where only hierarchy is significant), but is relevant in the Method Editor view (dealing with programming statements).

Editing or creating individual nodes with either of these two views is accomplished by means of node-type specific dialogs (generically termed node-type dialogs). These node-type dialogs collectively form a part of the SUI being proposed here. While each node-type dialog is unique, they have been designed so that a common SUI interaction paradigm is used, providing an orthogonality which aids in the learning and retention by a VASDE user.

3.2.1 Example Node-type Dialogs

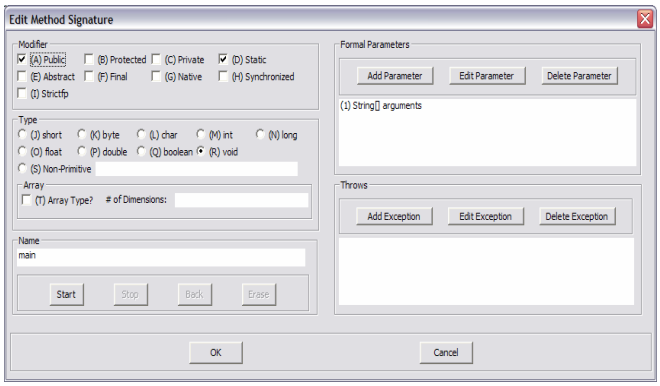

Figure 2 Create/Edit Method Signature Dialog.

Figure 2 depicts the node-type dialog used to create or edit a method signature. Standard GUI elements are used in the same manner as in standard Windows applications. The major difference is that the primary mode of interaction with these elements is via speech rather than via a mouse or keyboard. This approach keeps the user in a familiar environment, and provides familiar visual feedback. Some of the important elements in this dialog that are common to other dialogs are:

- The button labels are also "speakable" commands.
- Items in a group of check boxes (the Modifier group) are labeled with a unique character, and are selected and deselected by speaking "*select label*" and "*deselect label*", respectively.
- Items in a group of radio buttons (the Type group) are also labeled with a unique character (distinct from other groups). Items are selected with the "*select label*" command.
- Items in a list (the Throws and Formal Parameters lists) are labeled with a number. If a list item is to be selected from a dialog containing a single list, this is accomplished using the "*select label*" command form. In the Figure 2 dialog there are multiple lists with items that must be edited within subdialogs. Therefore, there are special commands (listed on the buttons) which are applied to a given list's items. In Figure 2 the *Add*, *Edit* and *Delete Exceptions* buttons are examples of such commands.
- A Name control is used to enter new names.
- Items on dialogs that must be filled in with Java expressions (other than new names) are entered by the keyboard on the current limited version of VASDE, and are indicated on the dialog by a standard text box.

The Name control allows users to create new identifier names for projects, classes, methods, fields, and variables, and is one of the few places that employs the dictation grammar of the recognizer. The operation is as follows:

- The user speaks "*Start*". This invokes a dictation-result listener; activates the *Back*, *Erase*, and *Stop* buttons; and deactivates the *Start* button.
- The user can now speak any word to begin creating a name, as long as the spoken word is not one of the other recognized dialog commands
- If the user speaks several words, they will be concatenated into a single word (e.g., "*MyProject*" or "*HelloWorld*"). The capitalization rules used in this concatenated word are context-dependent: lowerCamelCase is used for methods, fields, and variables, and UpperCamelCase is used for project, class, and interface names.
- If there is a mistake either by the user or by the recognizer, the user can speak "*back*" to erase just the last word spoken or "*erase*" to erase the entire word and start over.
- When the name has been entered to the user's desire, the user must speak "*stop*" to deactivate the naming control and dictation listener.

3.2.2 Example For-Loop Dialogs

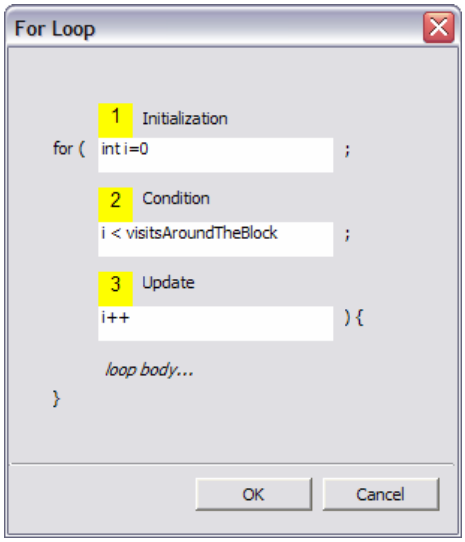

Figure 3 For Loop Dialog.

Figure 3 depicts the For-Loop dialog. This dialog is typical of the dialogs used as part of the Method Editor. Code "templates" such as this generate much of the "boilerplate" code – like keywords, parentheses, and semicolons – and only permit the user to edit the "changeable" parts of the template. In the case of the "For-Loop", only the "OK" and "Cancel" push buttons have equivalent voice commands. The three text boxes must be populated with Java expressions and, therefore, must currently be filled by using the mouse/keyboard interface. Ongoing research is extending VASDE to include a voice activated expression editor. The remainder of the Java syntax depicted in the dialog is automatically generated. Each of the supported Java structure types has its own unique dialog that allows the creation and editing of that structure.

By means of the speech-enabled node-type dialogs in conjunction with the two speech-enabled tree views discussed above, a Java programmer is able to create and edit a completely functional Java program in an environment that is voice-driven.

3.3 Usability Study

A small pilot evaluation was conducted to gather user feedback on the completeness, usability, and appropriateness of the VASDE interface for the task of Java programming. A small pool of evaluators was chosen from the ranks of university students and professional programmers. Although the user group that could benefit the most from the VASDE interface is programmers with manual disabilities, it was difficult to find many such users; however, one of the evaluators used did have a severe manual disability.

For each subject, the evaluation process began with training the speech recognition engine. The evaluator was then provided with a tutorial that introduced the major functions of VASDE. After the tutorial was completed, each evaluator was given a sequence of individual tasks that involved transcribing and editing a Java program (provided as standard Java source text) in the VASDE application. Finally, each evaluator was given a questionnaire that was used to provide feedback on VASDE.

The survey instrument had 20 questions: 16 were 5-point Likert scale questions, 3 were yes/no questions, and a final question provided space for general open-ended comments. Additionally, each question provided space for open-ended comments. Eight questions addressed the completeness of the interface. Eleven addressed the appropriateness of the interface to the task of Java programming. Of these eleven, five addressed the subcategory of usability of the interface, and the rest were aimed at determining the evaluator's satisfaction with the overall interface. An attempt was made to gauge how appropriate they felt that the interface was for the task of editing Java programs. This addressed both the syntax-directed and the speech-enabled aspects. The results of this survey are presented below.

3.3.1 Results

3.3.1.1 Questionnaire Results

Two sources of feedback used were (1) the questionnaire filled out by each evaluator, and (2) direct observation of the evaluators during the given tasks.

	Category	E1	E2	E3	E4	E5	Median
	Completeness						
C1	Overall Completeness	4	4	4	2	3	4.00
C2	Program Creation	4	5	4	4	4	4.00
C3	Add/Edit Class/Interface	4	5	5	4	4	4.00
C4	Add/Edit Class Field	4	5	4	4	4	4.00
C5	Add/Edit Method/Constructor	4	5	5	4	4	4.00
C6	Add/Edit Method Body Code Structures	4	5	4	4	4	4.00
C7	Code Navigation	4	5	3	3	4	4.00
C8	Program Compilation	5	4	5	3	3	4.00
	Usability						
U1	Overall Usability	3	4	4	4	3	4.00
U2	Efficiency	4	4	5	3	3	4.00
U3	Attractiveness	5	5	5	4	3	5.00
U4	Responsiveness	3	3	5	4	3	3.00
U5	Intuitiveness	4	4	5	5	4	4.00
	Appropriateness						
A1	Syntax-directed approach NOT viable for editing	2	1	1	2	2	2.00
A2	Syntax-directed approach LESS appropriate than tradi-	3	2	2	3	4	3.00

	tional editor with added speech recognizer						
A3	Limitations of speech recognizer make interface too frustrating to use	3	2	1	3	4	3.00
A4	Interface impedes user	Y	N	N	N	N	N/A
A5	Speech commands are appropriate	Y	Y	Y	B	Y	N/A
A6	Comfortable with editor for everyday use	N	Y	Y	Y	N	N/A

Legend:
1 – Strongly Disagree
2 – Disagree
3 – Neutral
4 – Agree
5 – Strongly Agree

Y – Yes
N – No
B – Both yes and no

For each of the Likert questions, the median answer was equal to, or higher than, the median achievable score (3 on a scale of 1 = least favorable, to 5 = most favorable). Although all of the scores were favorable for VASDE, the scores were higher for the categories of completeness and usability than for the general appropriateness of the interface.

While all of the feedback gained through the evaluations was valuable, it is likely that the most can be learned by concentrating on the areas where VASDE scored the lowest. Only three of the questions asked scored just at 3 on the Likert scale. These questions were:

1. Is the syntax-directed, voice-driven approach less appropriate than the combination of a traditional editor with an added speech recognizer? (A2 in table)
2. Do the limitations of the speech recognizer make the interface too frustrating to use? (A3 in table)
3. Rate the interface's responsiveness on a scale of 1 to 5 (U4 in table).

In the case of the first question, two reasons for the lower scores are immediately evident. Firstly, the strictly syntax-directed environment introduces a major interface paradigm shift for a programmer with even minimal experience with "standard" programming tools. Secondly, the very fact that the interface is tightly constrained could seem too inflexible to programmers who often enjoy the customization and personalization that many programming environments support.

The second and third questions may be the most important ones of the evaluation. In many ways, the entire design approach of VASDE is meant to make the speech recognition system as useful as possible. This question asks whether the VASDE interface has done enough in this area to overcome these limitations. The feedback indicates that there is still room for improvement. The fundamental cause of some of the low evaluations was however related to performance issues of the underlying speech engine. Thus, it is felt that the approach used in the VASDE system was "valid", but that a broader acceptance of the approach may first require improvements in those underlying support levels.

Although these three particular areas mentioned above reflected the most negative feedback from the questionnaire, the majority of the user feedback was quite positive. In fact, for all areas of interface completeness and all areas of interface usability save responsiveness, the median scores were 4 and above (on a scale of 1 to 5).

3.3.1.2 Observation results

In addition to direct user feedback, observations during the evaluation process also yielded fruit. A major observation was that the speech recognition engine still produced many errors, despite approximately one hour of voice training. Recognition errors also seemed to be greatly influenced by the "speed" of an evaluator's natural speech. Additionally, the engine even appeared to be sensitive to even slight changes in the speaker's voice, as observed for one particular evaluator whose cold constantly affected the tone of his voice during the evaluation.

The most frequent recognition error that was observed was a misrecognition. This occurred quite frequently when evaluators were using dialogs with character labels (for example, the "New Method" or "New Class" dialogs). In such cases, single characters were often mistaken for one another (e.g., 'A' and 'K'; 'D', 'E', and 'P').

A final observation from the evaluation process is that the evaluators did not seem to retain all of the speech commands after the tutorial session. Therefore, they resorted to consulting the User's Guide with some frequency during the non-tutorial tasks.

4. Conclusions

4.1 The technology

The behavior exhibited by the underlying support for VASDE (i.e., the speech engine and JSAPI code) was somewhat disappointing. Evaluators were in some cases frustrated by both performance and accuracy problems. Although performance was an issue, the promise of faster computers in the future can help to mitigate the effects of a slow recognition process. From evaluation comments and observations, the most glaring source of frustration is the lack of recognition accuracy.

The recognition accuracy might be increased in three identified ways. The first suggested approach is simply increasing the engine training time (training was limited to 40-60 minutes for these evaluations). The second possible approach to improved recognition could be to include extensive training on the command-words used by VASDE. Such training should reduce the misrecognitions based on the command words used. The third suggestion is to alter the actual speech commands themselves. A design decision was made to use very concise label names in the command + label spoken command form. Labels were usually a single number, a single character, or a combination of a character plus a number. While this approach was easy for the user to learn, it also presented many possibilities for misrecognition. For example, the "select A" command was quite frequently misrecognized as "select K". Observation of the evaluation process showed that commands which featured more words (for instance, "insert method declaration after B 1") were rarely misrecognized. It is likely that the longer commands provide more context for the speech recognition engine and can lead to greater accuracy. Unfortunately, longer commands are likely to be harder to remember by users.

One performance observation made during the evaluation process was that at times there was a time lag when user-spoken input was being ignored. This occurred most noticeably when a program context shift called for a change in the grammar. In such cases the programmer, unaware of the dead-time, would begin speaking prematurely and would have to repeat his commands. The remedy, used in subsequent changes to the VASDE implementation model, was to provide the user with a red-green light icon indicating when speech input was being accepted. This remedy proved to be successful [7].

4.2 The Interface

The frustrations experienced by testers based on the current weaknesses found in the lower levels of this project (i.e., the speech recognition engine and the JSAPI interface) make it somewhat difficult to isolate their attitudes about the interface itself. However, the survey taken seems to indicate that the evaluators looked favorably upon the completeness and the usability of the VASDE application despite these weaknesses.

In addition to the questionnaire described above, all evaluators were given an opportunity freely to provide feedback on VASDE. As it relates to one of the major goals of this research effort, some of the most valuable feedback involves the unique comments of the single evaluator with a severe manual disability. In general, the comments of this evaluator were very positive. His main suggestion was that there should be less typing (although he acknowledged that the addition of an expression editor would alleviate this concern). His most telling comment was that, even with the numerous recognition errors that occurred this interface was much faster for him than if he had had to enter items by hand. Therefore, he considered it "much better than having to type" and intimated that an interface like VASDE would make him much more productive for everyday programming work.

4.3 Future directions

The research efforts undertaken in the development of this voice-activated editor suggest many opportunities for future research.

VASDE only implements a portion of a total voice-activated programming solution. The first obvious extension to this research would be the design and implementation of a voice-activated Java expression editor. This research is currently underway.

Of course, VASDE could also be enhanced with more IDE-level features such as enhanced project management, a run-time debugging environment, integrated help, GUI building, and support for other programming languages besides Java. If all such extensions were adequately voice-enabled, the combination of all these features plus VASDE would result in a fully capable voice-enabled IDE.

Another area of future research suggested by VASDE is the exploration of which voice commands are most appropriate for a given interface. Specifically, such research could explore what characteristics of voice commands result in the best rate of recognition (and lowest rate of misrecognition) balanced with the highest rates of user retention and appropriateness to the given task. Another area under investigation for improving the overall success of VASDE is to implement the concept of synonym sets. These sets would be based on individual user input testing. For example, when a given programmer speaks the word "clear" the speech engine might routinely be hearing the word as "Claire". Thus, for that particular user treating "Claire" and "clear" as synonyms would reduce the number of cases of misrecognized keywords. This concept would of course require a more extensive training period.

The code used to implement VASDE is available from the authors.

5. References

[1] Arefi, Farah, Charles E. Hughes , and David A. Workman. "Automatically Generating Visual Syntax-Directed Editors." Communications of the ACM, Vol. 33, No. 3. ACM Press. New York, NY. 349-360. 1990.

[2] Biddle, Robert, Ewan Tempero, and Glen Wallace. "Smarter Cut-and-Paste for Programming Text Editors". *Proceedings of the 2nd Australasian Conference on User Interface.* IEEE Computer Society Press. Queensland, Australia. 56-63. 2001.

[3] Désilets, Alain. "Context Sensitive Magic Words for Programming by Voice". *Proceedings of the 1st VoiceCode Design Session.* Boston, MA. 2000. 21 August 2002. <http://voicecode.iit.nrc.ca/VoiceCode/VCode1stMeeting/Alain_Desilets/ CSMWs/index.htm >

[4] Désilets, Alain. "Miscellaneous Techniques for Programming-by-Voice". *Proceedings of the 1st VoiceCode Design Session.* Boston, MA. 2000. 21 August 2002. <http://voicecode.iit.nrc.ca/VoiceCode/VCode1stMeeting/Alain_Desilets/ misc_techniques/index.htm>

[5] "Eclipse.org Main Page". 5 March 2005. *The Eclipse Foundation.*< http://www.eclipse.org>

[6] Epstein, Jonathan. "Programming by Voice Using Continuous and Discrete Methods". *Proceedings of the 1st VoiceCode Design Session.* Boston, MA. 2000. 21 August 2002. <http://voicecode.iit.nrc.ca/VoiceCode/VCode1stMeeting/Jonathan_Epstein/index.htm>

[7] Froeseth, Julie. "A Model for Voice-Activated Visual GUI Editors", Masters thesis, University of South Alabama, December 2005.

[8] Heintzelman, Matt and Phil Pfeiffer. "Machines, statues, and people: strategies for promoting RSI awareness in computing curricula". *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*. San Jose, CA. 296-300. 1997.

[9] Hennessy, Sean. "Computer Code Voice Transcription". 6 January 2005. 7 March 2005. <http://www.h-dm.com/resources/HappyHands_Java/hh_paper_short_technical.html>

[10] Horwitz, Susan and Tim Teitelbaum. "Generating Editing Environments Based on Relations and Attributes." *ACM Transactions on Programming Languages and Systems*. New York, NY. 557-608, 1986.

[11] "Java Speech API Programmer's Guide". Version 1.0. 26 October 1998. *Sun Microsystems, Inc.*< http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/>

[12] Johansson, Eric. "Speech Driven Coding". 10 March 1998. 21 August 2002. < http://www.connact.com/~esj/voice_coding/voicecoding2.PPT>

[13] "Lost-worktime Injuries and Illnesses: Characteristics and Resulting Time Away from Work, 2000". Bureau of Labor Statistics. 10 April 2002. 12 October 2002. <http://www.bls.gov/news.release/osh2.nr0.htm>

[14] Oviatt, Sharon L., Philip R. Cohen, and Michelle Wang. "Toward Interface Design for Human Language Technology: Modality and Structure as Determinants of Linguistic Complexity." *Speech Communication, 15* (3-4). European Speech Communication Association. 283-300. 1994.

[15] Reiss, Steven P. "Pecan: Program Development Systems That Support Multiple Views." *Proceedings of the 7th International Conference on Software Engineering*. Orlando, FL. 324-333. 1984.

[16] Shmerling, Robert. "Harvard Commentary: Computer Use and Carpal Tunnel Syndrome". *InteliHealth*. 27 August 2001. 12 October 2002. <http://www.intelihealth.com/IH/ihtIH/WSIHW000/20813/20888/332014.html?d=dmtJHNewsArchive>

[17] Steindl, Christoph. "Benefits of a Data Flow-Aware Programming Environment". *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press. Toulouse, France. 105-109. 1999.

[18] *VoiceCode White Paper. National Research Council of Canada*. 23 February 2005. <http://voicecode.iit.nrc.ca/VoiceCode/public/wiki.cgi?VoiceCode_white_paper>