A voice-activated programming IDE for manually disabled programmers

D.D. LANGAN¹, T.F. HAIN¹, T.J. HUBBLE², & J. FRØSETH³

¹University of South Alabama, U.S.A., ²TAK Imaging, Mobile, AL, U.S.A., ³UNINETT, Trondheim, Norway

(Accepted date  )

Abstract
Purpose: The paper presents the design and implementation of voice-based interface to support two types of application development activities (coding, and visual GUI design) by manually disabled programmers.
Method: We implemented two prototype programs to implement and test each of the two proposed voice-based interface models.
Results and Conclusions: The interface design concepts, and the results and lessons learned of the usability studies are discussed. The usability studies indicated that the interface models were sound, although the voice recognition aspects left something to be desired.

1. INTRODUCTION

Application development for programs having graphical user interfaces (GUIs) typically involves (1) the use of an editor for coding purposes, and (2) the use of a tool to visually layout and specify attributes for forms and widgets. Numerous commercial tools exist that integrate these two application development subtasks (e.g., Visual Studio), wherein coding is typically a typing-based activity, and visually developing graphical user interfaces (GUIs) relies heavily on the mouse. While neither of these is problematic for most programmers, both may prove difficult for those with various manual disabilities. Hand and arm movements required by such interfaces may be difficult, painful, or impossible for programmers with physical disabilities such as carpal tunnel syndrome. Programmers, due to their prolonged and heavy computer usage, are susceptible to a family of ailments known as repetitive strain injuries (RSI), defined as “a soft-tissue disorder that results from the repetitive use of some part of the body”[6]. These are serious conditions that can lead to numbness, pain, and, in extreme cases, paralysis[6]. Also supporting the need for better programming tools for the manually disabled, are statistics that show that students with orthopedic impairments are more likely than those without to major in the computer and information sciences[8].

For the manually disabled, voice input is an obvious alternative to the combination of keyboard and mouse. Some existing voice-based products for the manually disabled are implemented by a simple voice-activated adaptation layer being placed on top of an existing application.

This research looks at the common subtasks of code entry and GUI layout and investigates their nature with the intent of developing and testing models of how they should be performed by voice. The combination of models for these subtasks form the components of a Speech User Interface (SUI) intended for application development by the manually handicapped.
2. CODING BY VOICE

Most accessibility-enabled applications (e.g., email and word processors) take a pre-existing interface, and simply layer voice support over it. This approach misses the opportunity that an interface designed from the outset for voice may be more optimal.

The speech interface requirements for disabled programmers are different from those of disabled users of other applications. For example, 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”), a variety of punctuation marks, and a relatively wide variety of user-defined identifiers not found in any natural language (e.g., “myPointer”). For new identifiers it is necessary to resort to a ‘spelling interface’ of the voice system. Clearly, in order to make speech recognition a viable input method for programmers, some approach other than dictation is needed.

2.1. Objectives

The intent of providing an alternative interface for coding by manually disabled programmers was to create a programming editor that met the following objectives:

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, edit, and navigate code quickly.
4. Effective, in the sense of reducing the probability of programmer-generated syntax errors.

We wanted to evaluate whether a speech interface coupled with a syntax-directed editing approach would provide a “natural” match. 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 enumerated by the IDE, and selected vocally by the programmer. In many instances the possibilities are singular, in which case the IDE 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). To provide a usability-testing environment for this SUI concept, a voice-activated syntax-directed editor (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 might be possible. Objective 2 was ensured by staying as close to standard interface conventions as possible. 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 from lists is less error prone and requires smaller vocal bandwidth.

2.2. Research Related to PBV

2.2.1. SUI Research

In the general area of SUI research, Oviatt[15] 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, she found that users preferred the constrained interfaces by a two to one ratio.

2.2.2. Syntax-directed Editing

In the area of syntax-directed editing environments, Horwitz [9] presents a design for a language-independent model of editing involving the representation of programs as abstract-syntax trees with an associated relational database. Arefi [3] further expands upon the basic tree view by specifying languages as directed, labeled graphs, which allowed for quick updates to the program structure. Reiss [16], 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 [4] 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 further proposes a selection mechanism that relies on program units (such as the name of an object or a method), rather than arbitrary characters. This research project took advantage of these earlier results. In parallel with the current work, Hennessy [7] describes an independent commercial effort named “Happy Hands Java Speech Editor,” developed to solve the same problem as VASDE. While VASDE and Happy Hands share a similar syntax-directed, tem-
plate-based SUI approach, they are realized in very different ways in the respective UIs. By comparison to Happy Hands, VASDE’s approach is unique in the areas of tree-based navigation and selection as well as the ability to create new names by voice.

2.2.3. Programming by Voice

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

2.2.4. Speech Recognition Technology

Sun Microsystems [2] provides practical guidelines for the implementation of speech technology in general applications, and addresses key areas like performance and the choice of a proper spoken command set.

2.3. An Overview of VASDE

VASDE is a voice-activated syntax-directed editor for Java programs. Because it was intended as a research tool rather than as a commercial IDE, a few limiting design decisions were made, including the omission of a few linguistic features, and the omission of a voice-activated expression editor. We are currently exploring the topic of using voice for the low-level task of expression editing.

The underlying platform for the VASDE implementation is the Eclipse [1] open source project, which provides a wide set of useful libraries, frameworks, and applications. VASDE is designed as an application plug-in to the Eclipse environment, and borrows heavily from its infrastructure.

Another existing technology, the Java Speech API (JSAPI)[2], provides the interface between VASDE and the underlying speech recognition engine, and provides the means for recognizing speech and triggering required actions. It also allows the specification of a “grammar” of speech commands that would be matched by the speech engine.

2.4. VASDE Details

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

There are four components in the IDE:
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 refers to the main application shell (consisting of the main window, the menu bar, and the tool bar) and hosts the other UI components. 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 with a separate user and speech interface. A small set of voice-enabled elements were added to the Application Framework to allow it to open projects, build the path required by the compiler and exit the program.

![Figure 1 VASDE Main Application Interface.](image)

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 and navigation of a program takes place by means of some syntactic element of the program rather than a less meaningful unit such as a line of text. While the organization and selection of syntactic units takes place directly within these tree views, editing is accomplished by means of a collection 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, class/package 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 is used to apply a single command to a sequential group of labeled nodes. Most of the commands are applicable to any node types, although some types recognize additional context-sensitive commands. The set of commands includes the following spoken words: (a) “select”, (b) “expand”, (c) “contract”, (d) “add” node type—[Project Explorer], (e) “insert” node type (“before” | “after”) [Method Editor], (f) “delete”, (g) “copy”, (h) “cut”, (i) “paste” [Project Explorer], (j) “paste into”—[Project Explorer], (k) “paste” (“before” | “after”) [Method Editor], (l) “edit”, (m) “edit (method | constructor) signature”—used in Project Explorer; to invoke the Method Editor

Node-type specific dialogs are used to edit or create individual nodes within each of these two views. The collection of node-type specific dialogs forms a part of the SUI being proposed here. While each dialog is unique, they have been designed so that a common SUI interaction paradigm is used, providing an interface orthogonality which aids in the learning and retention of VASDE.

2.4.1. Example Node-type Dialogs

Figure 2 depicts the node-type dialog used to create or edit a method signature. Standard GUI elements are used with the major difference being that the primary mode of interaction is by speech rather than by mouse or keyboard. This approach keeps the user in a familiar environment, and provides visual feedback. Some of the important elements in this dialog that are common to other VASDE dialogs are:
1. The button labels are also “speakable” commands.
2. Items in a group of check boxes (e.g., the Modifier group) and items in a group of radio buttons (e.g. the Type group) are labeled with a unique character, and are selected and deselected by speaking “select label” and “deselect label” (for checkboxes), respectively.
3. Items in a list (e.g., a Formal Parameters list) 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, e.g., Add, Edit, and Delete Exceptions.
4. A Name control is used to enter new names.
5. Items on dialogs that must be filled in with Java expressions (other than new names) are currently entered by the keyboard in the present limited version of VASDE, and are indicated on the dialog by a standard text box.

![Create/Edit Method Signature Dialog](image)

The Name control allows users to create new identifier names for projects, classes, methods, fields, and variables. This dictation-mode control is entered by speaking “Start” and is exited using “Stop”. “Back” can be used to remove the last word spoken, and “Erase” can be used to remove the entire spoken input. If the user speaks several words they will be combined to make a single identifier (e.g., “MyProject”).

2.4.2. Example For-Loop Dialogs

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—such as 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. 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.

![Figure 3 For Loop Dialog.](image)

### 2.5. Usability Study

A small pilot evaluation was conducted to gather user feedback on the completeness, usability, and appropriateness of the VASDE interface. A small pool of evaluators was chosen from the ranks of university students and professional programmers. One of the evaluators used in the pilot study 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. Finally, each evaluator filled out a survey.

The survey had 16 five-point Likert scale questions, three yes/no questions and an open-ended question. Eight of the questions addressed the completeness of the interface (e.g., to do a particular task), five dealt with the usability of the interface (e.g., to evaluate responsiveness) and six were used to assess the appropriateness of VASDE to the task of programming (e.g., level of frustration caused by use of voice). The complete presentation of the survey results can be found in [10].

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 to 5). Although all of the scores were favorable for VASDE, the scores were better 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. Three of the evaluation items scored just 3, and were:

1. The syntax-directed, voice-driven approach is less appropriate than the combination of a traditional editor with an added speech recognizer.
2. The limitations of the speech recognizer make the interface too frustrating to use.
3. Rate the interface’s responsiveness on a scale of 1 to 5.

Two reasons for the lower scores on the first question are immediately evident. Firstly, the strictly syntax-directed environment introduces a major interface paradigm shift. Secondly, the fact that the interface is tightly constrained could seem too inflexible to programmers who often enjoy the customization and personalization of their programming environments.

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, although the fundamental cause of some of the low evaluations was related to the performance of the underlying speech engine. While it is felt that the approach used in the VASDE system was deemed valid, broader acceptance of the approach may require improvements in the underlying support levels.

Although these three questions reflected the most negative feedback from the survey, the majority of the feedback was quite positive. In fact, for all areas of interface completeness and all areas of interface usability except responsiveness, the median scores were 4 and above.

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 almost an hour of voice training. Recognition errors also seemed to be greatly influenced by the speed of an evaluator’s natural speech. The engine 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 frequently observed recognition error was misrecognition. This occurred quite frequently when evaluators were using dialogs with character labels. In such cases, single characters were often mistaken for one another (e.g., ‘A’ and ‘K’).
3. GUI DEVELOPMENT BY VOICE

The visual creation of GUIs differs in a number of significant ways from the more general task of programming, and can be broken down into several smaller tasks including:
1. the selection of a widget to add to a window,
2. the positioning/sizing of that widget on the window,
3. the setting of desired widget properties, and
4. the specification of event-handlers, which is best handled by voice using a strategy like the one in VASDE.

Positioning and sizing of widgets in a visual environment for non-disabled programmers is typically achieved using a mouse. Malkewitz states that voice is not well suited for moving a mouse pointer on a screen, nor is it intuitive to the user [14]. While this may not be the most effective means of doing visual GUI development, it may be the best option for manually disabled programmers.

What follows is a description of a model to allow voice for GUI development, and a description of a tool created to test the basic tenets of that model. The results of a pilot study to evaluate this approach to GUI development are presented.

3.1. Voice-Activated GUI Editor Model

The purpose of the GUI Editor research model was to suggest a general framework that could be used in the development of a voice-activated GUI editor addressing the four fundamental tasks listed above. Initial studies led us to certain concepts that were central to the design of a high-quality voice-activated GUI editor, and we used them to form the basis of our research model. A voice-activated GUI editor should:
1. exhibit usability, i.e. it should be easy to learn in addition to being easy to use,
2. allow for spatial issues to be addressed in a natural way,
3. minimize the vocal load on the user’s voice,
4. contain incremental and reversible actions,
5. provide visual feedback to the user,
6. be efficient, i.e. require a minimum number of steps to perform a specific task,
7. draw on “user recognition” rather than “user recollection”,
8. not implement an overly strict scoping strategy, and
9. prevent the user from making mistakes.

3.2. Voice-Activated GUI Editing Research

Voice over Workplace (VoWP) provides a speech interface for one particular application, namely SAP Workplace [11]. The objective of this research was to evaluate the trade-offs between ambiguity handling and efficiency in addition to observing how the different paradigms affected the users. The study showed that users preferred relaxed scoping (wherein users could navigate more freely) to rigid scoping (wherein more precise navigation was required), even though the former caused more ambiguity. The reason given was that navigation was easier when involving fewer commands. The study also showed that implicit scoping was preferred over relaxed scoping, in part because it reduced the occurrence of ambiguity.

Malkewitz maintains multimodal applications require special attention in certain areas [14]. His application allowed users to utilize a pre-existing painting program, and, using an overlaid interface, navigate by using voice and head gestures. He identified three main practical aspects that should be present in multimodal systems. First, the software should be “application-independent”; i.e., to avoid user frustration, any initiated dialog should allow the user to exit at practically any time. Second, the software should have “application-specific comfort”; i.e., should be powerful and comfortable to use. Third, the software needs to exhibit “user tailorability”; i.e., the user can customize the application to suit his or her needs. The author emphasized that the latter aspect is especially important for users with disabilities.

Kamel experimented with a drawing tool intended for visually impaired people [13]. His tool replaced mouse-based interfaces with voice navigation combined with a 3×3 grid interface. The 3×3 grid provided a way for users to navigate the screen in a “divide-and-conquer” style. Nine cells are not nearly enough to be useful, so to permit finer granularity, the application allowed users to subdivide each cell into another 3×3 grid, and one more time, resulting in a 27×27 grid. Any finer granularity than this turned out to be problematic for the users. The division of the computer screen into a 3×3 grid was found to be intuitive to users.

3.3. The Implementation

Like VASDE, VAGUE was created as a research vehicle, and implemented support for a number of widget types sufficiently representative to test the model; buttons, labels, text-fields, combo-boxes, and group-boxes. Also like VSADE, VAGUE was constructed as an Eclipse platform plug-in and generated Java code to implement the described GUI. A fuller description of this implementation can be found in [5].

The VAGUE system is composed of three major visual elements: (a) an editor called GUI Editor, (b) a view called Voice Input, and (c) a perspective called Voice-Activated Java.

Initially, the user is presented with a blank window onto which widgets will be placed as editing proceeds. The GUI Editor is the portion of the screen that shows the state of that window. The user may specify a title and other key attributes for the window itself. The GUI
editor can be toggled to show a grid with cell sizes of 50 pixels or 100 pixels. Using voice commands, this grid can be used to specify an initial position for widgets as they are added to the GUI.

Figure 4 shows the Voice Input View. This view is used to convey to the user the current mode of the program, the voice commands available in that mode, the last command “heard” by the program, and the status of the microphone (i.e., paused or ready).

Figure 4 The Voice Input view

Dialogs are used extensively in VAGUE to specify command arguments, to allow widget property specification, and to present error messages. Figure 5 shows the widget-selection dialog invoked when the user has issued the “New” command from the main editing mode. The user can issue the command, for example, “New Button” to bypass this dialog.

Figure 5 The New Widget dialog

This dialog is typical of all dialogs used in VAGUE in that it allows the user to use either the numerical label (e.g., “3”) or the textual version (e.g., “Text Field”) and has its own context-specific Voice Input. Another example of a dialog is where the user speaks “select” in the main editing mode to select an existing widget, and can choose from a numbered list of existing widgets.

The dialog shown in Figure 6 is used to specify the properties associated with a GUI button. This dialog again shows a consistent style, and illustrates that in some cases the user enters a special dictation mode to create longer phrases (e.g., to specify the variable name). In the Voice Input portion, the reserved words “finish” or “OK” are used to exit the dictation mode.

Figure 6 The Button Properties dialog

As suggested above, VAGUE has different editing modes. These modes reflect the task at hand and impact the currently available command set. Figure 7 shows a state transition diagram for the editing modes used in VAGUE.
There are certain actions a user expects from a voice-activated GUI editor. These can be grouped into five top-level types of commands. A listing of these with their designated scope is presented in Table 1.

As shown in Figure 7, there is a relatively small number of total commands (about 70), with very few available at any given moment. This small vocabulary makes VAGUE easier to learn and master.

<table>
<thead>
<tr>
<th>Command type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Universal commands</td>
<td>Can be invoked almost anywhere in the GUI Editor. Example: “pause” to stop voice input.</td>
</tr>
<tr>
<td>2 Widget commands</td>
<td>Can be applied to one or more widgets. Certain commands are widget specific. Example: “delete” to delete a widget.</td>
</tr>
<tr>
<td>3 Navigation commands</td>
<td>Can be invoked when moving one or more widgets from one place to another. Example: “right” to move a widget right.</td>
</tr>
<tr>
<td>4 General commands</td>
<td>Can be invoked anywhere as long as no dialogs are open. Example: “exit” to exit the GUI Editor.</td>
</tr>
<tr>
<td>5 Miscellaneous commands</td>
<td>Each command has its own scope. Example: when in Scroll mode, “scroll down” to scroll the form down.</td>
</tr>
</tbody>
</table>

### 3.4. Usability Study

Using a number of volunteers, an experiment—comprised of a fixed set of tasks representing typical visual GUI development—was conducted to evaluate the proposed model. The hardware used consisted of a laptop and a headset for the recording of the test subjects’ voice input. Eclipse version 3.0.2 provided the framework for VAGUE. Voice-related software included IBM ViaVoice Pro USB Version for Windows, Release 9.0 and CloudGarden’s TalkingJava SDK and Java Speech API implementation, version 1.6.3.

Six subjects participated in the pilot study. Prior to the experiments, test subjects had to train the voice engine. This training included reading one of three 10-15 minute stories that are part of the ViaVoice software. In addition, subjects trained a specific set of frequently used words, including “ok”, “finish”, “insert”, “resume”, “pause”, “next”, and “properties”.

The subjects were presented a tutorial on the use of VAGUE prior to the actual testing. They were permitted to ask questions during the tutorial, but not during the testing.

After completion of the experiment, subjects were asked to complete a survey about their experience. The survey included 22 Likert 5-point scale (1 = strongly negative to 5 = strongly positive) closed questions (each with an option to include a comment), and three open-ended written questions. The survey questions assessed (a) the satisfaction with individual editor operations, (b) the overall responsiveness and look and feel of the editor and (c) their attitude about using voice for the given tasks. The complete set of results from this survey may be found in [5]. The averages for the 22 closed questions ranged from 2.5 to 4.83. Only 5 of the questions had an average at or below 3.5. The responses to the survey indicate that the test subjects were satisfied with the editor as a whole. This is supported by answers to questions which asked about test subjects’ opinion about the efficiency of the editor, using voice input, and the user-friendliness of the editor’s interface. Furthermore, test subjects also showed satisfaction with specific elements such as visual clues and actions like adding and moving widgets.

The major cause for frustration was the misrecognition of words. In the future, this can be partially alleviated by letting users customize commands. In situations like the naming of widgets where any word must be recognizable, the only current options are more training of the voice engine, or a more responsive and accurate voice engine.

Even though users were fairly positive in their assessment of VAGUE, they were less positive when it came to questions about using voice input to design GUIs. Answers to questions asking the subjects if they found the new input mode workable for the tasks that they had been required to perform showed some of the lower scores. A discussion of the three open-ended question responses follows.

**Question 23:** Was there anything about VAGUE that you particularly appreciated?

Some subjects commented on the editor holistically, for example two subjects said it was “easy to use.” Others mentioned specific features that stood out to them. For example, that it was “easy to move between forms”, that the subject liked “the way that [the current mode] was displayed and the list of active commands”, and that it was helpful “being able to see the last recognized word”.

**Question 24:** Was there anything about VAGUE that you particularly disliked?

Several subjects pointed out that it was frustrating that some words were not picked up very well by the voice engine. Most other remarks were concerning
features the subjects felt the editor lacked, but did not come as a direct dissatisfaction with any of the editor’s current features. Such remarks included that it would be valuable to somehow indicate the currently selected object, and that it would be beneficial to list several of the previous commands rather than just the most recent one.

Question 25: Do you have any other comments regarding this survey or VAGUE?

Subjects tended to end the survey on a positive note. Example comments included: “Very interesting experience. Should only get better as technology progresses.”, “It was very nicely done. I thought it was very usable.”, and “This was fun to do.”

4. CONCLUSIONS

4.1. The Technology

The behavior exhibited by the underlying support for VASDE and VAGUE (i.e., the speech engine and JSAPI code) was somewhat disappointing. In some cases, evaluators were frustrated by both performance and accuracy. In the future, faster computers will help mitigate the effects of slow word recognition.

The recognition accuracy might be increased in three identified ways: increase the engine training time, include more extensive training on the used command-words, and alter the actual speech commands themselves. For example, a design decision was made in VASDE 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”. In general, the use of numbers (used extensively in VAGUE) tended to cause fewer misrecognition errors than the use of letters. Observation of the evaluation process showed that commands that featured more words (for instance, “insert method declaration after B 1”) were rarely misrecognized. It is likely that the longer commands provide more context and redundancy, and thus greater accuracy. Unfortunately, longer commands are likely to be harder for users to remember.

One performance observation made during the VASDE evaluation process was that there was sometimes 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 very successful 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.

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 surveys seem to indicate that the evaluators looked favorably on the completeness and the usability of both the VASDE and VAGUE applications despite these weaknesses.

With respect to the major goals of this research effort, some of the most valuable feedback came from the single evaluator who had a severe manual disability. In general, his comments 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, 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.” He intimated that the VASDE and VAGUE interfaces would make him much more productive for everyday programming work.

5. FUTURE DIRECTIONS

VASDE and VAGUE together implement only a portion of a total voice-activated solution for program development for manually disabled programmers. Both programs made simplifying assumptions and limited the features available, but could be enhanced to offer a fuller range of features. Another obvious extension to this research would be the design and implementation of a voice-activated expression editor to be incorporated into VASDE, and this work is currently underway.

An area of research suggested by these projects 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 recognition rate and accuracy, balanced with the highest user retention and appropriateness to the given task. Another area under investigation for improving the overall success of VASDE and VAGUE is to implement the concepts of
synonym and homonym sets 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 homonyms would reduce the number of cases of misrecognized keywords. Similarly, synonym sets would give the user more flexibility in their choice of keywords to use (e.g., setting “quit” as a synonym for “cancel”).

6. References