Computer and Information Science
www.ccsenet.org/journal.html
Vol. 1, No. 1
February 2008

A Model for Voice-activated Expression Editing

David D. Langan
School Computer & Information Sciences, University of South Alabama
307 University Blvd N, Mobile, AL 36688, USA
Tel: 1-251-460-6390 E-mail: david.langan@gmail.com
Thomas F. Hain
School Computer & Information Sciences, University of South Alabama
307 University Blvd N, Mobile, AL 36688, USA
Tel: 1-251-460-6390 E-mail: thain@usouthal.edu
W. Christopher Camery
Lockheed Martin Simulation, Training, and Support,
12506 Lake Underhill Rd., Orlando, FL 32825, USA
E-mail: chris.camery@gmail.com

Abstract

Code entry and editing by manually disabled computer programmers is difficult in conventional GUI-based development environments, since these rely heavily on mouse and keyboard use. The advent of accurate and responsive speech recognition technologies has made the speech user interface (SUI) a viable option for input. While high level programming constructs can be entered in a fairly natural way us/ing voice, expressions may be very complex in structure, and may use a wide variety of variables, literals, punctuation marks and operator symbols. A SUI-based syntax-directed editor was previously presented, but its interface model did not include expression editing in its model and implementation. This paper fills this gap by presenting a SUI-based model for entering and editing programming language expressions. A Java implementation was evaluated by a group of programmers to verify the functionality and to test the ease of use of the model. The results of this evaluation are presented and discussed.

Keywords: Speech user interface, Expression editor, Integrated development environment, Accessibility

1. Introduction

For programmers, entering and editing code in a development environment involves the manual operation of input devices such as the keyboard and mouse. This provides a daunting, if not impossible challenge for people with manual disabilities ranging from repetitive strain injuries (RSI) to more severe disabilities. A 2006 report from the Bureau of Labor Statistics (USDL, 2007). reported that those who suffered from carpal syndrome experienced a median of 27 days of missed work. These statistics provide evidence that programmers other than just the manually handicapped could profit from a form of input other than the mouse and keyboard. The relatively recent advent of accurate and responsive speech recognition technologies has made speech user interfaces (SUIs) a viable option for program entry. A SUI-based development environment offers accessibility to programming for manually disabled programmers. There is also the possibility that if programming by voice is sufficiently intuitive and fast, users who are not disabled may at times choose this option if the programming environment is multi-modal. What is needed for SUIs is a "standard" for their use such as now exists for GUIs.

1.1 Editing Code by Voice

The term *expression* in the context of expression editing is used to mean a collection of variables, function calls, and operators that are combined in a logical manner according to a language specification. Expressions are syntactic elements of programming languages found, for example, as the predicate of *if*, *while* or *for* statements, as right-hand sides of assignment statements, or as function parameters. Expressions, more than most other constructs of programming languages, can be quite long and intricate.

Here we will discuss three existing models for voice-driven programming: Voice Code (NRCC, 2005c), Happy Hands (Hennessey, 2005b), and Voice-Activated Syntax-Directed Editor (VASDE) (Hubble, 2006 and Hubble 2007). Voice

Code and Happy Hands integrate expression editing into the overall coding framework, using a dictation mode. As will be shown, there are distinct inadequacies in this approach. VASDE takes the view that expression editing is inherently different from producing program code, being at a much higher level of abstraction, and defers expression editing altogether (provisionally using GUI input in the interim). This paper presents a model to fill this gap.

1.1.1 Voice Code

While entering text by voice may work well for dictating natural language documents, the Voice Code documentation (NRCC, 2005a) contains a very good example of why dictating code is so difficult, and uses an expression to demonstrate this point: “Programming languages were never meant to be spoken. Consequently, it is very difficult to dictate code.

For example, to dictate the following C++ code:

```
if (currRecNum < maxOffSet)
{
    ^
}
```

the programmer might have to say something like this:

if open-paren Charlie uniform romeo romeo cap romeo echo Charlie cap-November uniform mike less-than maxbegin-capitalize begin-no-space off set end-capitalize end-no-space close-paren new-line open-brace new-line new-line close-brace up-arrow tab-key” (NRCC, 2005a).

The Voice Code team decided on a more natural-language approach to dictating the above expression, in this example they proposed something like “*if current record number is less than max offset do the following”* (NRCC, 2005a). The Voice Code project treats expressions in a very general way. The only support for expressions in Voice Code are simple tools for inserting and navigating general balanced expressions. Commands for inserting expressions include the following:

between <exprName>

<exprName> pair

empty <exprName> (NRCC, 2005b)

The semantics of these commands are explained as follows:

The *"between <exprName>"* and *"<exprName> pair"* forms are used to type the balanced expression and put the cursor inside the expression. The *"empty <exprName>"* form types the balanced expression but puts the cursor right after it. (NRCC, 2005b).

For navigating, there are commands such as “*jump out*” and “*jump back out*” which place the cursor just outside of the expression currently being edited. As Voice Code is designed to deal with any programming language, it does not include any form of code assistance (such as Microsoft’s Intellisense) for any particular language. Although Voice Code currently supports only Python as of November 2006, support for other languages, including C++, Java, and Perl is planned to be added in the future. Voice Code does not replace the GUI paradigm, but rather augments it. Voice Code is built as an extension to the text editor Emacs, and thus is merely a speech layer built atop a traditional GUI (NRCC, 2005c).

1.1.2 Happy Hands

Sean Hennessey created the Happy Hands Java Speech Editor (Hennessey, 2005b). This editor deals only with Java as the editing language, and therefore has Java-specific features. Happy Hands is of interest due to its similarities with Hubbell’s VASDE, both of which were developed concurrently. The similarity of Happy Hands to VASDE, makes it a relevant starting point to examine in the design of an expression editor (Hennessey, 2005b).

The expression editor of Happy Hands, called the “expression transcriber”, allows the creation of expressions by reading, from left to right, the elements that make up the expression. It is also possible to specify that an operator will be to the left of the current variable. However, Happy Hands does not provide a way to remove a part of an expression. The Happy Hands documentation itself admits “Happy Hands does not have a good way to remove elements from expressions, other than using the backspace key. Since the expression is quickly re-parsed and active transcribers are updated, this is perfectly valid the result is no different.” (sic) (Hennessey, 2005a).

While in Happy Hands, the user must use the mouse or keyboard to select the element he or she wishes to edit, in VASDE the selection of an existing expression to edit is done entirely by voice.

According to the Happy Hands documentation, expressions are implemented as “for the most part binary trees, with some lists too” (Hennessey, 2005a). Whenever an expression element is being edited the existing element is overwritten unless an operator is spoken, in which case new nodes will be added to the binary tree, one for each side of the binary operator.

While both Happy Hands and Voice Code make it easy to enter expressions by voice, editing them is entirely too difficult.

1.1.3 VASDE

Hubbell (Hubble, 2006 and Hubble 2007) presented a model and implementation called VASDE (Voice Activated Syntax-Directed Editor) to create and edit Java programs. He takes the position that entering code provides an opportunity for semi-automation by the fact that computer languages have a rigid syntax for high level constructs. The syntax-directed approach used by Hubbell uses a different paradigm than conventional text editing. Instead of typing a code block or statement, there are separate editors for each type of statement or block. For example, to create a for-loop, a dialog will appear and the user is required to enter the initial condition, the condition for continuation, and post-loop action into text fields. The code for the loop is then generated. Using this paradigm, programmers create and edit structures and statements rather than editing text directly.

The tests and evaluation of the VASDE editor yielded favorable results (Hubble, 2006 and Hubble 2007), and showed the viability of both the syntax-driven approach, as well as the use of voice for the task of programming. However, the low level task of expression creation and editing was bypassed in the initial research. It was felt that expressions were too “small” and complicated to be naturally supported by the language syntax driven approach that had been appropriate for higher level constructs.

2. Proposed Model

Expressions, across a spectrum of modern high level programming languages, tend to be more alike than different. The Voice Expression Editor (VEE) model proposed here is based on Java syntax, but can be easily ported to other high-level object-oriented programming languages. The Java Language Specification (JLS) (Gosling, 2000) states that expressions are composed of syntactic elements, including variables, operators, casts, literals, class names, method invocations, and subexpressions.

The VEE-model proposed here enforces the restrictions imposed by the JLS (e.g., two binary operators may not be adjacent). However, the editing process does not structurally conform to the formal JLS definition of an expression wherein each node in the syntax tree is an individual syntactic element (Shavor, 2003). For example, the simple expression **`3*4`** consists of a three-node tree with the root being **`*`**, and its children being **`3`** and **`4`**. While this syntax description lends itself nicely to the parsing and code generation, it proves to be awkward as the basis for by-voice expression editing, leading to unnecessarily long navigation paths. VEE “flattens” this syntax tree, creating a subexpression only when a term is surrounded by parentheses, or is an element of a list (e.g., an argument to a method having multiple arguments). VEE would view the expression, **`3*4`** as a single node containing three elements. The individual elements within a node are contained within a “slot.” In the previous example, the expression would have 3 slots. Elements are dynamically allocated to the slots from left-to-right.

Expression editing involves several identifiable sub-tasks, namely (a) navigation, (b) selection/insertion/deletion, and (c) data entry. The sections that follow will discuss each of these in detail.

2.1 The Navigation Process

When an expression is being edited by voice, it is appropriate to add visual clues to facilitate the navigation process, such as the automatic numbering of all slots, and the highlighting of slots that are currently selected. A special expression-editor—distinct from the program-editor—is invoked for this purpose. The two editors communicate in the sense that the program-editor, when required, invokes the expression-editor (potentially pre-loading an existing expression), and the expression-editor when terminated returns control to the program-editor, passing it the created or edited expression.

The expression-editor will recursively invoke itself when a subexpression (i.e., a subnode in the VEE syntax tree) needs to be edited, initially loaded with that subexpression. In each case, a command-word such as *done* or *OK* is used to terminate an instance of the expression-editor to return to the calling context. This relationship may be seen in the left side of the Figure 1. The role of the clipboard is explained below.

2.2 Selection/Insertion/Deletion

Expressions are composed of elements, each of which is dynamically (from left-to-right) allocated to a slot. A slot may contain one of the following element types:

a keyword (e.g., **`this`**, **`true`**, **`false`**, **`new`**)

an operator,

a literal,

an identifier (variable, or type),

an expression within parentheses (other than a parameter list), and

an element of a list (elements separated by '**,**' or '**;**').

The last two element types represent "subexpressions".

We first discuss the selection, insertion, and deletion of expression elements in general terms, and then discuss the insertion of each of the specific element types.

2.2.1 Selection

With a mouse or keyboard, one can click on, or arrow to, any desired element within an expression prior to performing some desired operation. In the VEE-model, we allow direct access to elements within an expression by the use of the automatically generated and displayed slot numbers. The numbers are ordered from left to right.

In the expression **`(y+1)+3*add(2*x,8)`**, there are 7 slots (numbered 1 to 7), containing **`y+1, +, 3, *, add, 2*x,`** and **`8`**. The displayed slot numbers enable selection of a slot by saying *select* and then the slot number (e.g., *select 4* to select the slot containing **`*`**), or a range of slots by saying *select A to B* (e.g., *select 2 to 3*).

In this model, the editing process is hierarchical in the sense that selecting a subexpression leads to invoking a new instance of the editor. In the above example, the slot **`y+1`** would be edited within its own editor. One would say *select 1* to select the slot, and then say *edit* to recursively enter a new instance of the expression-editor (subexpression-editor) to work on just **`y+1`**.

2.2.2 Insertion

All expression elements are inserted immediately after the currently selected slot. Slot zero is always present at the beginning of any expression (even an empty one), although it has no visible content. Therefore, to add something to the beginning of an expression, slot zero should be selected.

If the expression resulting from an insertion would be invalid, a blank slot will be automatically inserted. For example, if the expression being edited is **`2+4`** (with 3 slots), and the slot containing **`2`** is selected, and the user inserts a **`-`** operator (by saying *minus*), then the resulting expression will be **`2-_+4`** (having 5 slots) because two binary operators cannot appear adjacent to one another. Here, the expression editor creates blank numbered slots containing a placeholder, **`_`**.

2.2.3 Deletion

To delete a slot, or a range of slots, the user would say *select* to pick the desired slot(s) and then say *delete*. The slots are immediately removed from the expression and are placed in the recent-clipboard (discussed below).

If a deletion would create an invalid expression, blank slots (each containing a placeholder, **`_`**) will be inserted as needed. For example, if the user were to delete the **`+`** operator from **`1+2`**, the resulting expression would be **`1_2`**, with slot 1 as the selected slot.

2.3 Insertion/Editing of Specific Element Types

2.3.1 Insertion of keywords and operators

The insertion of keywords (after the selected slot) is accomplished by simply saying the keyword, e.g., *null*, *false*, *true*, *this*. The insertion of both unary and binary, logical and numeric operators are similarly invoked by words which are recognized by the speech engine and accepted by the editor. For example, to insert **`+`**, one would say *plus*. A sampling of other recognized words are *minus*, *preincrement*, *postdecrement*, *times* (or *multiply*), *divide, logical-and, binary-and*, *mod* (or *modulus*).

The expression **`1+2*3`** can be entered by saying *one plus two times three*, pausing between each word until the "recognizer busy" indicator (discussed below) disappears.

2.3.2 Literals

Numeric literals are inserted by speaking the number that is to appear in the expression. For integers the actual number must be spoken. For example, if the user wanted to enter the number 136751, then they would say *one hundred thirty six thousand seven hundred fifty one*. In the case of a float like 3.14 the user would say *three point one four*. Character and String constants can be entered by saying *character* or *string* to enter a special entry mode. In the case of *string*, the user enters a dictation mode that he exists by saying *done*.

2.3.3 Subexpressions

To create a subexpression, the user can select a slot or range of slots and say *subexpression* or *parentheses*. The selected slot or slots will be converted into a single subexpression slot, surrounded by parentheses. If the current expression is **`3+2*x`** (having 5 slots), and the user wanted **`3+2`** to be a subexpression, then the user would say *select one through* *three* to select the appropriate slots, and then say *subexpression*. The resulting expression would be **`(3+2)*x`,** (consisting of three slots).

When a slot containing a subexpression is selected, a new invocation of the expression-editor is generated when the user says *edit*. This editor is preloaded with the subexpression (minus the parentheses, if it is a parenthesized subexpression), and can be edited as an independent unit (with its own set of slots.) Once the *OK* command is given, this (sub)expression-editor is terminated, and in the invoking editor, the original subexpression is replaced by the edited subexpression. This process can be repeated recursively for expressions with nested parentheses, as shown in Figure.

2.4 Insertion of Identifiers

This topic is sufficiently complex to warrant a separate section on its own. While numeric literals and binary operators can be entered by speaking the name of the number or the operator, identifiers cannot be dictated in this manner. Variables, methods, classes, and packages are not guaranteed to be English words, or even to be pronounceable. We make use of the fact that the VASDE program-editor requires identifiers to be declared before use, and that these identifiers, within a given scope, can therefore be known by the expression-editor. Identifier entry into expressions is done by selection rather than dictation. The challenge, therefore, was to find a method whereby identifiers are presented to the programmer in a navigable manner, so that the choices at any given time are not overwhelming. We call the mechanism we use a "clipboard" window.

2.4.1 The Clipboard

The clipboard window is used to enter text into expressions. If the text entries are package names, class names, method names, or variable names, a spoken command (e.g., *package*, *imports*, *class*, *method*, *variable*) is issued to generate and display a context-sensitive numbered list of applicable entries—reminiscent of "intellisense". Loading the clipboard can be facilitated by the interaction of VEE with the program-editor to determine the in-scope items. The item to be entered into the expression is selected and inserted by saying the associated number. This approach minimizes required vocal bandwidth, and the possibility of speech misrecognition. While this obviates the need for mouse and keyboard (a necessary goal), it incidentally provides a memory aid for the programmer. Also, by comparison, this method of choice avoids the possibility of misspelled names (a common problem with the keyboard).

The clipboard also forms the by-voice replacement of the ubiquitous cut-and-paste clipboard found on editors utilizing mouse/keyboard interaction. If the spoken command is *recent*, a list of previously selected fragments (individual slots, or slot ranges) that were deleted or copied from an expression are listed, and can be pasted into the current expression.

We can think of the clipboard as having different versions, depending on the invoking command, i.e., the type of entry to be inserted into the expression. Some versions may invoke additional versions, for example selecting a method may then require selecting a class, etc. The clipboard intelligently navigates such hierarchical structures, as will be shown below. Figure 2 shows a more detailed view of how the clipboard would work internally and relative to the expression (subexpression) editor for access to names.

Other available clipboard navigation commands include *back* and *cancel*. The *back* command does what one would assume: it takes the user back one step, effectively undoing the previous command. The *cancel* command immediately closes the clipboard and returns control to the expression editor without modifying the expression.

We now give details of the clipboard versions, and their interactions. Initially we describe the selection process for each version, and then talk about insertion into expressions.

2.4.2 Variables

If the user wants to enter a (a) local variable, (b) instance variable (possibly inherited), (c) class-variable (possibly inherited), or (d) particular data-field of an object, one would enter the clipboard by saying *variable*. The clipboard is now titled "Variable Clipboard," and is put into a state of "Accepting Variables." Because variables can be instances of a class, the user must be able to select not only the variable itself, but any available fields of that variable. If a variable contains accessible fields, that variable will appear in the clipboard in two forms: the variable name and the variable name followed by a dot (e.g., **`x`** and **`x.`**). If we assume that **`x`** is a local variable that has an available data member **`y`**, the user would see a menu that offered both **`x`** and **`x.`**. If he now selects **`x`** from the menu and says *OK*, he would return to the expression-editor with an **`x`**. On the other hand, if he selects **`x.`** he would see a new menu of available variables that would include **`y`** which he could then select to return **`x.y`** to the expression editor. This navigation applies recursively to fields of fields, etc.

2.4.3 Methods:

If the user wants to insert a method call, the user would enter the clipboard by saying *method*. The clipboard is now titled "Method Clipboard," and is put into a state of "Accepting Methods." The clipboard would display a numbered list of all methods (possibly inherited), including overloads, available in the current context, i.e., the current class. For example, if the enclosing class has two (possibly inherited, but visible) methods with signature **`thePerson(INT)`**, and **`thePerson(INT,INT)`**, both will appear in the clipboard numbered list.

If the user wants a method from a different class, or one that can be invoked by a variable, they would say the appropriate command (*class* or *variable*). This would load the clipboard with identifiers associated with that class or variable. Thus, the clipboard can at different times offer available package names, class names, variable/field names or method invocations (with an appropriate formal parameter list). When a clipboard is titled as "Method Clipboard," what is meant is not necessarily "a clipboard that is currently displaying methods," but rather "a clipboard whose task is to allow the user to select a method."

An example may help to illustrate the creation of a method call. If the user wants to enter the method call **`x.f(2)`**, where **`x`** is a local variable, the user would say *method* to invoke the method clipboard. Upon entry, the clipboard would offer the instance and class methods appropriate to the calling context (this might not include **`f`**). The desired method is an invocation by a local variable, so the user next says *variable* to view a list of variables (which would include the local variables.) The user would then be offered **`x`**, and after he selects it and says *OK*, he would now see a list of methods available to **`x`**. This list would include **`f(INT)`** and possibly other versions of **`f`** (if it had been overloaded.) After selecting **`f(INT)`** and saying *OK*, the programmer would be returned to the expression-editor with `x.f(INT)` inserted in the appropriate slot.

Sometimes, the desired method will require a fully qualified name that includes the package name. Hence one may say *package* to select a package name from among those available. While packages in Java cannot contain other packages, they can have names that give the appearance of nested packages (e.g., **`javax.swing`** and **`javax.sound`** may appear to both be contained in the **`javax`** package). Packages with such names may have very long names, and there are numerous packages in the Java standard library. To make them more manageable, and to avoid showing several hundred packages to the user at once, the clipboard treats packages as if they were nested. For example, to select the package **`javax.swing`**, the user will first select **`javax.`** and then select **`swing`**.

The net effect of the clipboard nesting variables, classes, and packages, is that a user may select a particular item based on its fully-qualified name. For example, to select the method **`javax.swing.JOptionPane.showInputDialog(OBJECT)`**, the user will first say *method* to invoke the method clipboard. The user will then say *package* to view a list of packages. He will select the partial package name **`javax.`** , say *OK*, and then select the package itself **`swing,`** and say *OK* again. A list of classes belonging to the javax.swing package will now be displayed. The user will then select the class `JOptionPane`, whereupon its methods will be displayed, and the user will select the method **`showInputDialog(OBJECT)`**. The entire command sequence required to enter the fully qualified method in the example would be something like : *method*, *package*, *8, OK, 4, OK, 90, OK, 21*" (where *8* is the menu-number corresponding to **`javax.`**, *4* is the number corresponding to **`swing`**, *90* refers to **`JOptionPane`** and *21* refers to **`showInputDialog(OBJECT)`**).

To prevent the user from having to select each part of a package name every time it is to be used, the clipboard keeps track of packages used and adds them to an "imports" clipboard, which can be accessed by saying *imports*. While this only saves one selection in the case of **`javax.swing`**, the user might be spared several selections in the case of the longer names.

2.4.4 Classes

For subexpressions used in casts, a type (class) is required, and the user would enter the clipboard by saying *class*. The clipboard is put into a state of "Accepting Classes". The clipboard will list both inner classes, and their dotted versions (if necessary). The behavior after selecting an item is much like that of variables and methods. The mechanism for full qualification of classes is like that described for methods.

3. VEE an Implementation

The VASDE project focused on programming in "the-large" and did not provide a mechanism for the voice input of expressions. The VEE model was proposed to rectify this missing component. The VEE implementation (VASDE Expression Editor) was written in Java as an Eclipse (Budinsky, 2004) plug-in so that it would function with VASDE. VASDE was modified to use VEE anytime a user needed to enter or edit an expression, and VEE when it terminated would return the expression to VASDE.

The VEE implementation followed the model presented in the previous section. The expression-editor was implemented using a dialog as shown in Figure. Here one sees the stack nature of the expression editor. In this case, the user entered the editor with the expression **`10+(x+(18-4))*2`** (which has 5 slots). The user then selected slot 3, said *edit*, and entered the editing of the subexpression **`x+(18-4) (`**which has 3 slots). The user then selected slot 3, **`18-4 (`**also having 3 slots,) and requested an *edit*. Finally he selected slot 3 of this subexpression, where he could use *delete* to remove the **`4`** and then use some other command (e.g., *five*) to insert a new element, **`5`**.

The upper portion of the expression editor shows the stack, while beneath that, the current (sub)expression is shown with its slot numbers, and the selected slot(s) being highlighted. After entering/editing a subexpression one may say *OK* to return to the previous subexpression in the stack, with changes being reflected in that expression. Finally, the last *OK* would return the programmer to the program-editor, with the final edited expression.

The VEE clipboard is called upon from the expression editor and is used to enter variables, methods, class names, package names, and recently used text. That clipboard shows a numbered collection of items that may be selected as described in the previous section (see Figure 4).

4. Evaluation

Five people with experience in C++ or Java were selected to test and evaluate the effectiveness of VEE as a voice-driven expression editor (Camery, 2006). The total evaluation took about three hours. The first phase of the evaluation required the evaluators to use the voice-training features of the voice-recognition engine (in our case IBM Via-Voice version 9) to train it to recognize their speech patterns. In the second phase, the evaluators were asked to dictate and edit given expressions into the SpeechPad utility program that was supplied with IBM Via-Voice. It was not required that the users complete this task, but only that they understood what would be involved. The purpose of this was to have them simply experiment with the expression editing task based only on dictation. In phase 3, the evaluators were exposed to a guided tutorial on the use of VASDE and all of the major features of VEE. During this time they were free to ask questions. After the tutorial was completed they entered phase 4, wherein they were asked to perform a series of VEE tasks on their own. The evaluation task was designed to use all features of VEE, and to allow the evaluator to see what it would be like to use the editor to create a program. In this phase, the evaluator was free to ask for help using VASDE (since they were not asked to evaluate VASDE itself,) but they were asked to complete the VEE task without assistance. This task took about an hour. Finally, in phase 5, the evaluators were asked to fill out an evaluation questionnaire regarding their thoughts on, and experiences with, VEE. The questionnaire included five 5-point Likert-scale questions regarding the evaluator's opinion of various aspects of the editor. At the end of the questionnaire, the evaluators were invited to make any additional comments regarding various issues with the model or its implementation.

The questions elicited the evaluator's overall opinion of the editor, the effectiveness of the use of slots, the effectiveness of the editor compared with alternative methods of voice input, and the degree to which the recognition problems of the underlying speech engine affected the evaluator's opinion of VEE.

The results of the 5-point Likert-scale (wherein 1 represents the least support for the statement, and 5 represents the strongest degree of agreement with the statement) questions are given in Table 1.

4.1 Evaluator Response to Ease of Use (Q1/Q5)

While the overall ease of use of the editor was rated poorly, every evaluator responded that the poor performance of the speech engine limited the effectiveness of the editor, and led to frustrations while using VEE. The most common problem was that the speech engine misrecognized words. This was not found to be a problem for most of the main screen's functions (with the exception of entering numeric literals). The clipboard screen was found to have the most misrecognition errors, but these were again related to numbers. The recognizer would too often misrecognize a number being spoken (e.g. the number 190 being recognized as 119). This clearly frustrated several evaluators. One evaluator commented that the speech engine was "a definite weak point in the process".

The rate at which some evaluators initially attempted to enter data in the main editor window was too fast for the speech engine to recognize. Once the evaluators began to speak more slowly, there were far fewer misrecognition errors. The red "recognizer busy" indicator was of great help to these people.

One evaluator stated that the ability to say the names of methods and variables to select them from the clipboard would be nice, as opposed to selecting them by number. Another evaluator expressed the desire to be able to declare a new variable while inside of the expression editor.

It should be noted that these tests were conducted using the IBM Via-Voice (version 9). Subsequent to these tests, the latest version of Dragon Naturally Speaking was tested for voice input and was found to be both faster and considerably more accurate. In particular it handled the input of numbers far better than IBM Via-Voice.

4.2 VEE Compared to Simple Dictation (Q2/Q4)

Every evaluator stated that they would rather use VEE than use a traditional text editor with a set of voice macros. Several evaluators noted that while entering an expression in VEE may take longer, editing expressions was fairly painless.

However, no evaluator seemed particularly enthusiastic about using voice to enter expressions, and in fact, question 4 received the lowest score of all the questions. The explanation for this, however, may lie in the universal opinion that the underlying voice engine used was largely at fault (question 5).

The responses from all evaluators indicate that while keyboard input is preferable to voice input for entering and manipulating expressions, if they *had* to use voice input, they would want to use something like VEE. This observation tends to support the main goal of this project, namely to provide a good voice-activated editor for the manually disabled. None of the five evaluators was manually disabled. It is possible that a programmer with such a disability would have an entirely different perspective.

4.3 Evaluator Response to the Use of Slots (Q3)

It is interesting that, while most evaluators responded fairly negatively regarding the use of voice as an viable means of entering expressions, they rated the effectiveness of the use of slots quite highly. Several evaluators commented that they would like to have the ability to break expressions into slots when using a standard text editor in order to more easily visualize and manage complex expressions.

In the open-ended portion of the evaluation, the evaluators stated that the use of slots is of great benefit to editing expressions by voice. One evaluator noted that it organizes expressions by syntactic elements, which is more appropriate for programmers than a simple text editor that uses only characters and whitespace. Another evaluator commented that "if corrections are necessary, the use of slots makes it easy to go back and edit things individually". It was concluded that the use of slots appears to be an effective method of organizing expressions.

5. Results and Conclusions

The results gathered here clearly reflect that, while the VEE implementation has some problems, these problems were based on the use of a poor speech engine, and not on the model itself. Later experiments showed that much better performance was available using the Dragon Naturally Speaking (version 9) speech engine. It is quite likely that the evaluators would have experienced a significantly lower level of frustration had this engine been used.

The VEE model for voice expression-editing is somewhat complex in its approach to the task, but the task itself seems to be inherently difficult to "explain" using words. One possible explanation for the weak evaluation of VEE may be that the evaluation process was too short thus not allowing the programmers to master the paradigm required to use VEE as compared to using a keyboard. It is speculated that with additional practice programmers would become more proficient in its use.

The authors and evaluators found the slot-approach to be fairly intuitive for the task for expression editing. This suggests that, while a voice-activated expression editor based on VEE might not be attractive to the general population of programmers, it might very nicely fit into a voice-activated editor (such as VASDE) for manually disabled programmers for whom the use of a keyboard and/or mouse is problematic.

References

Budinsky, Frank (2004). *The Eclipse Modeling Framework*, Addison Wesley

Camery, W. Christopher (December 2006). VEE: The VASDE Expression Editor. *Master's Project*, University of South Alabama, Mobile, AL.

Gosling et. al. (2000). *Java Language Specification*, Second Edition. http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html. Sun Microsystems.

Hennessey, Sean (2005a). Using the Happy Hands Java Speech Editor. http://www.hdm.com/resources/HappyHands_Java/manual/index.html.

Hennessey, Sean (2005b). The Happy Hands Java Speech Editor. http://www.hdm.com/resources/HappyHands_Java/manual/index.html.

Hubbell, T. J., Langan, D. , and Hain, T. F. (2006). A Voice-activated Syntax-directed Editor for Manually Disabled Programmers, *Eighth International ACM SIGACESS Conference on Computers and Accessibility*, Portland, Oregon, pages 205–212.

Hubbell, Thomas (2005). Voice-Activated Syntax-Directed Editing. *Master's Thesis*, University of South Alabama, Mobile, AL.

NRCC (2005a). Why code dictation is so hard. http://voicecode.iit.nrc.ca/VoiceCode/public/wiki.cgi?Why_code_dictation_is_so_hard. National Research Council of Canada.

NRCC (2005b). Dictating General Balanced Expressions. http://voicecode.iit.nrc.ca/VCode_1_Doc/public/wiki.cgi?obj = DictatingGeneralBalancedExpressions. National Research Council of Canada.

NRCC (2005c). Voice Code. http://voicecode.iit.nrc.ca/VoiceCode/public/ywiki.cgi. National Research Council of Canada.

Shavor, et. al. (2003). *The Java™ Developer's Guide to Eclipse*. Addison Wesley.

USDL (2007). Nonfatal Occupational Injuries and Illnesses Requiring Days Away From Work, 2006. *Bureau of Labor Statistics*. http://www.bls.gov/news.release/osh2.nr0.htm.

Table 1. Table of Evaluation Questionnaire

	E1	E2	E3	E4	E5	Mean
Q1 – Overall VEE was easy to use.	3	2	4	3	2	2.8
Q2 – VEE was easy to use compared to dictation.	5	5	4	4	4	4.4
Q3 – The use of slots was effective for the given task.	5	4	5	4	4	4.4
Q4 – Voice was as an effective means of entering expressions.	2	2	3	3	3	2.6
Q5 – Issues with speech engine limited the effectiveness of the VEE editor.	5	5	5	5	5	5.0

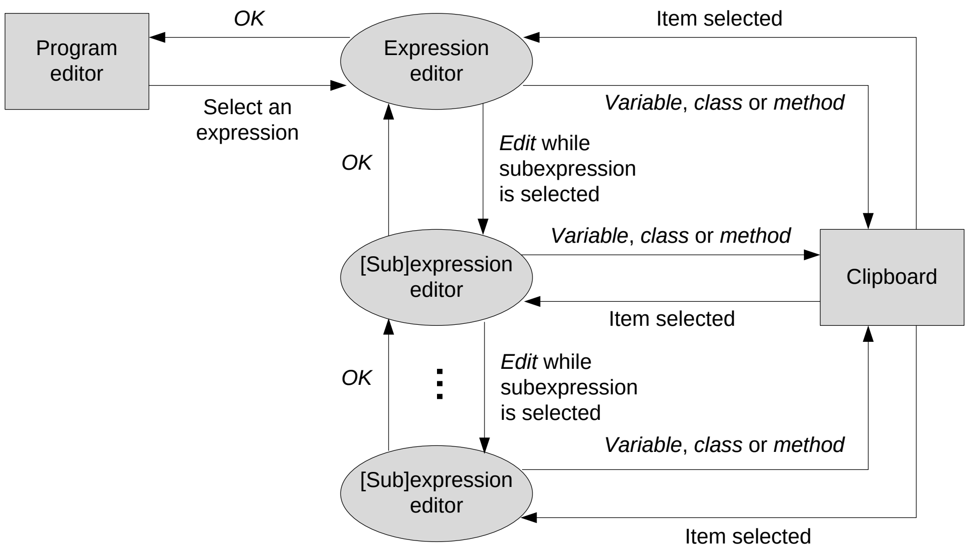

OK
Item selected
Program editor
Expression editor
Select an expression
Variable, *class* or *method*
Edit while subexpression is selected
OK
Variable, *class* or *method*
[Sub]expression editor
Clipboard
Item selected
OK
Edit while subexpression is selected
Variable, *class* or *method*
[Sub]expression editor
Item selected

Figure 1. VEE Control Flow Model

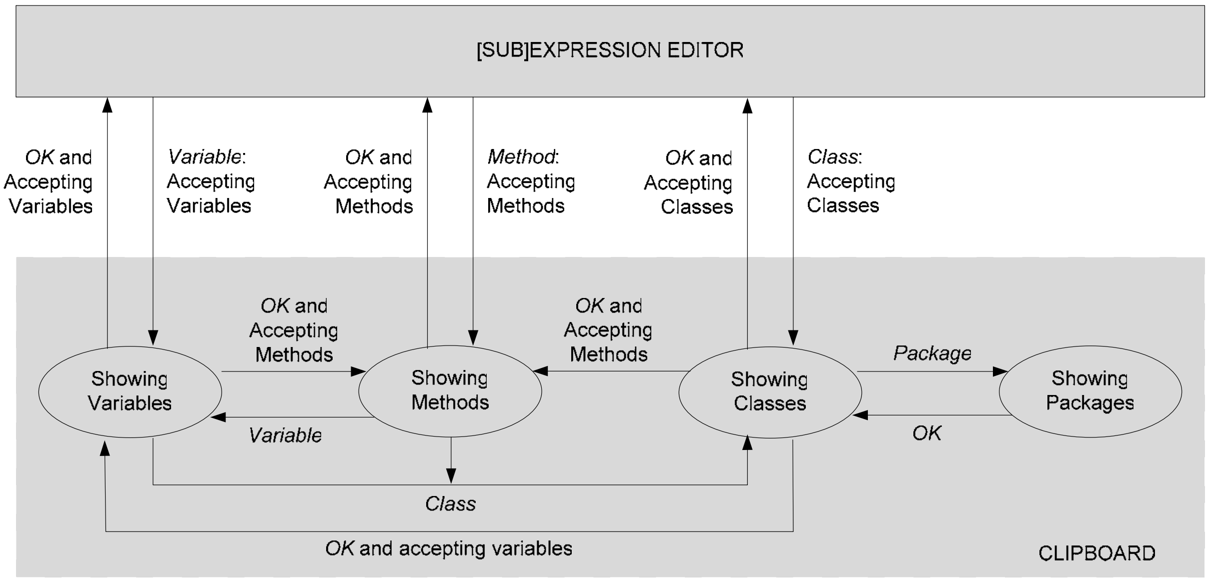
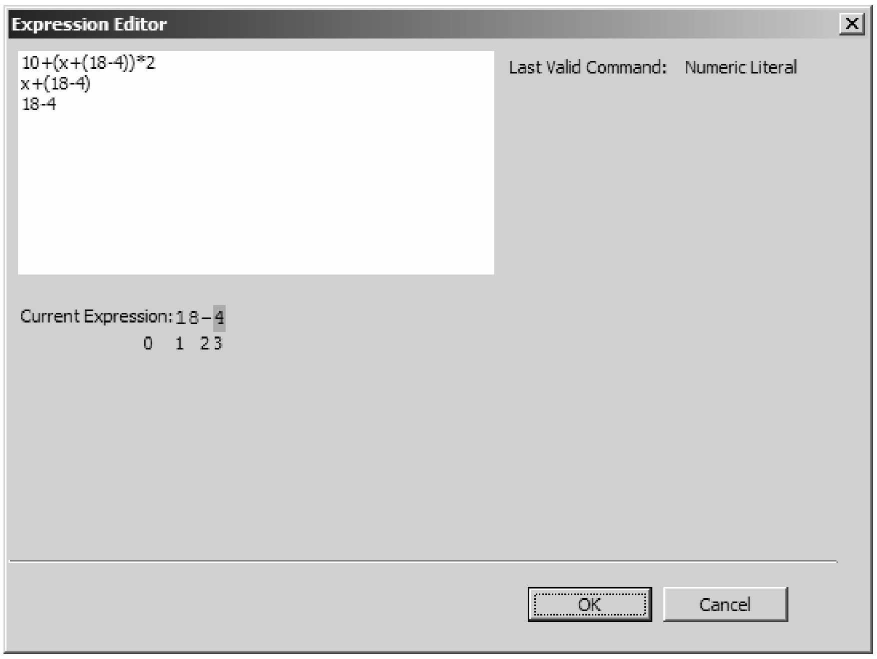

[SUB]EXPRESSION EDITOR
OK and Accepting Variables
Variable: Accepting Variables
OK and Accepting Methods
Method: Accepting Methods
OK and Accepting Classes
Class: Accepting Classes
OK and Accepting Methods
OK and Accepting Methods
Package
Showing Variables
Showing Methods
Showing Classes
Showing Packages
Variable
OK
Class
OK and accepting variables
CLIPBOARD

Figure 2. Clipboard States for Variable, Method and Class Entry

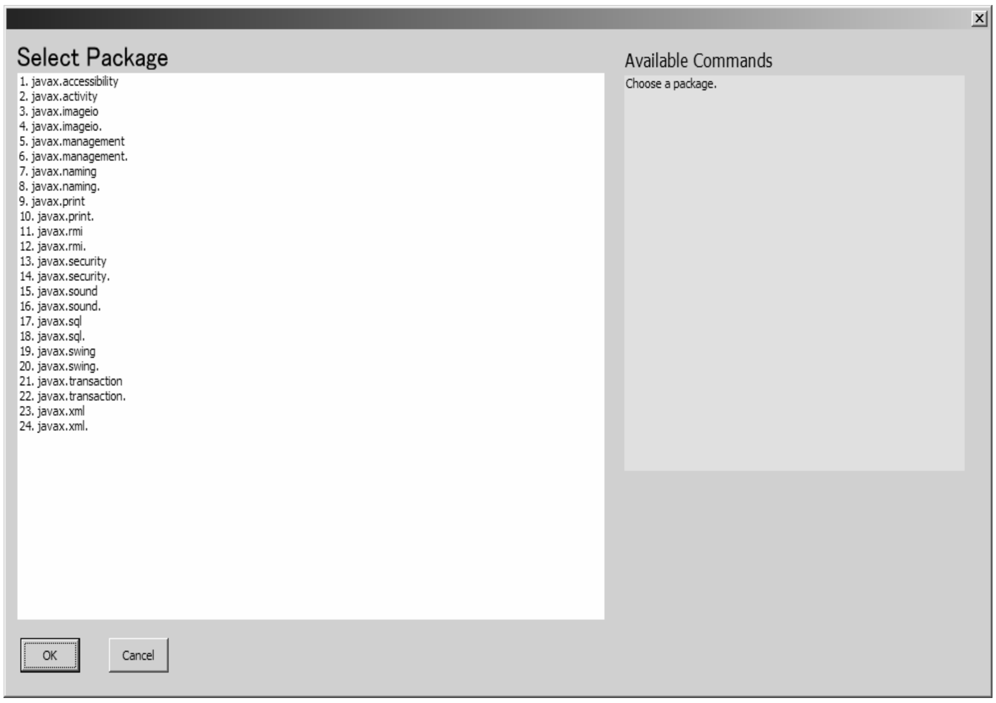


```
10+(x+(18-4))*2
x+(18-4)
18-4
```

Last Valid Command: Numeric Literal

```
Current Expression: 18-4
                    0 1 23
```

Figure 3. Screen Shot of Expression Editor



Select Package

```
1. javax.accessibility
2. javax.activity
3. javax.imageio
4. javax.imageio.
5. javax.management
6. javax.management.
7. javax.naming
8. javax.naming.
9. javax.print
10. javax.print.
11. javax.rmi
12. javax.rmi.
13. javax.security
14. javax.security.
15. javax.sound
16. javax.sound.
17. javax.sql
18. javax.sql.
19. javax.swing
20. javax.swing.
21. javax.transaction
22. javax.transaction.
23. javax.xml
24. javax.xml.
```

Available Commands

Choose a package.

Figure 4. Screen Shot of Package Clipboard