AN INVESTIGATION INTO PROGRAMMING BY VOICE AND DEVELOPMENT OF A TOOLKIT FOR WRITING VOICE CONTROLLED APPLICATIONS

Lindsey Snell
Department of Computing,
Imperial College of Science, Technology and Medicine, London.
Project Supervisor: Dr Elizabeth Pollitzer
Second Marker: Mr Jim Cunningham

15th June 2000

Abstract

This project investigates the problems associated with, and solutions to, the task of programming by voice. In addition it provides a toolkit for rapid development of voice−controlled software.

At present very little research has been carried out in the field of programming by voice but, as voice recognition software continues to improve and basic entry−level computers are able to cope with the computational demands of such software, interest is increasing. In the USA alone there are in the region of 3 million people who are unable to use a computer via normal methods but could interact with one by voice. Enabling such people to programme would open the door to the fastest growing industry in the world. So far research has been on informal basis (no papers have been written on the subject) and has focused on developing macros for existing text editors on the Unix platform.

This project aims to broaden the current state of research by developing an editor specifically designed to be controlled by voice, which supports the programming task and runs on all the major platforms. It also introduces new ideas for improving the programming task and provides a formal investigation into whether these methods (both those currently used and those proposed during the project) actually improve the task.

It was found that providing a method of automatically replacing certain keywords with specific text was the major factor in improving the programming task. Another key factor that improves efficiency is to provide a method of automatically creating variable, class and function names from a series of distinct words. Software has been written building upon approaches suggested by other research but the fastest method was found to be one developed by the author. This new method works by examining the context in which the text is dictated and automatically detecting when the user is trying to dictate a variable, class or function name.

The use of the software developed as a toolkit for rapid development of voice−controlled software was proven by developing a simple drawing package in a matter of hours.

Acknowledgements

First and foremost I would like to thank those who were involved in the HCI evaluation and without whom this project would not have been possible:
David Pearce, Selina Pavan, Simon Sheffield, Alex Millington, Lee Denison, Sarah Ewen, Graham Rowbottom, Jonathan Smith and Emmanuel Zolotas.

As well as giving up their time they were also eager to offer advice and their ideas broadened my view of the issues involved in programming by voice. They also provided an invaluable insight into the psychology of both the programming task and using speech recognition.

Many thanks also to my project supervisor, Elizabeth Pollitzer, whose idea the project originally was and also to Alain Désilets, founder of the VoiceGrip project in Canada and a leading light in research into programming by voice.

Contents

1. Introduction

This project investigates ways in which the task of programming by voice can be made possible and evaluates the effectiveness of the different methods proposed and the usefulness of the software developed. The project provides a set of tools to enable programming by voice as well as insights into writing software that is controlled by voice and the psychology of the programming task.

As voice recognition has gradually come out of the laboratory and into the public domain it has been heralded as a ground breaking tool for giving access to computers to those who are unable to use a computer by conventional methods but do have control of their voice. So far voice recognition software has concentrated on enabling people to dictate text to a computer, thus allowing them to write letters and e−mails and work on spreadsheets. While this has opened up a whole new area to such people there are still many other avenues that are less well explored, one of which is enabling people to write computer code by voice. This could allow people to create their own web sites or write programs for their own enjoyment or, more seriously, allow them to get a job in the IT sector, currently one of the fastest growing and best paid areas of employment.

While computer code can be dictated using voice recognition software and a text editor, it is a time consuming and tedious process. A couple of small research groups have developed ways of improving the task and, as interest in programming by voice has increased, these original ideas have started to develop into fully−fledged products. However these ideas have all focused on using macros with a Unix−based editor to speed up programming and no formal investigation has been conducted to see which of these methods improves the task the most, or indeed if it actually improves the task in terms of speed at all. The Unix−based editor is not designed in any special way to be used with speech recognition.

The aims of this project are twofold. Firstly to develop a voice−controlled editor, the CodeTalker, that allows rapid development of computer code by voice and which works with any voice recognition software on any platform (Unix, Windows or Linux). The features of this editor are decided by the second aim − to conduct a formal Human−Computer Interaction evaluation into which methods best improve the programming task and what features a voice based editor for programming should provide.

To determine the best methods to improve the programming task a set of three editors was developed, each of which solved the programming task in a different way. A formal Human−Computer Interaction Evaluation was performed using a group of ten programmers. Although not disabled themselves they were not allowed to use the keyboard or mouse at any time during the evaluation. From the results of this evaluation and the users own opinions an optimal editor was designed and evaluated. In addition a prototype editor that solved the problem in a radically different way was developed and evaluated alongside the other versions.

Why is this project original?

The software developed within this project looks at methods proposed by existing research groups as well as ideas of my own. Although some methods considered are the same as those developed by others it is important to note that there are a number of ways in which the software that I've developed differs from that currently available:

- Provides an editor purposely designed for the task rather than writing macros for an existing editor
- Rather than writing text and then invoking a parser to translate the text to code at some later point the text is translated as it spoken. It is the translated code that is displayed to the user, not the intermediate text.
- CodeTalker is designed to run on all major platforms (Unix, Windows and Linux) rather than being designed for Unix.
- Rather than requiring the user to have auxiliary software installed such as Perl and Emacs the CodeTalker is a self−contained package, the only additional software that the user needs is a voice recognition package such as Dragon Dictate.
- Rather than just try to enhance the programming task the CodeTalker has also tried to address the problems of designing a user interface to be controlled by voice rather than a keyboard and mouse.
- The prototype developed towards the end of the project provides a completely different method to solving the problem of programming by voice than anything developed so far.
- The software developed during this project provides a toolkit for developing all kinds of voice controlled software applications, not just ones to support programming.

The project does not aim to write or improve upon any of the existing voice recognition software, merely to harness the functionality that it provides to perform a specific task. The accuracy of any voice recognition is therefore a factor of the voice recognition software being used and not the CodeTalker software.

How to read this report.

There are three groups of people to whom this report is relevant, those interesting in developing software to enable programming by voice, those wishing to use the toolkit to develop other voice controlled software and those interested in the Human−Computer Interaction side of the project. The relevant chapters for each of these groups are:

1. Developers of tools for programming by voice

- Chapter 2 − gives an introduction to the current state of voice recognition technology, currently available tools for software developers wishing to program by voice and an overall idea of the number of people that would find such tools useful
- Chapter 3 – analyses the programming task itself and the looks at the specific problems related to programming by voice

- Chapter 4.2 & 4.7 – Describes the different approaches to solving the problems and describes the problems that are encountered when interacting with voice recognition software.
- Chapter 5.4 – Gives a detailed description the features were found to be the most use in enhancing the task and that should be included in any programming by voice software.

2. Users of the toolkit

   Those wishing to use the software as a toolkit to develop other voice−controlled software should refer to Chapter 4. The relevant sections are:

   - Section 4.3 – description of speech−directed interface that the toolbox uses.
   - Section 4.5 – gives reasons for why the software was developed in Java and links to Java programming resources
   - Section 4.6 − details the overall architecture of the toolkit given and explains the way in which the main classes interact.
   - Section 4.8 & 4.9 – describes how the software functions as a toolkit and gives a detailed explanation of how to add to, and adapt, it's functionality.

3. Those interested in the HCI Investigation

- Chapter 3 – analyses the programming task itself and the looks at the specific problems related to programming by voice
- Sections 4.2, 4.3 and 4.4 – introduce the different methods and software versions referred to in the analysis.
- Chapter 5 – contains the methods and full results from the keystroke analysis, user evaluations and expert evaluation. The results are fully analysed and the improvements seen and usability of the software developed discussed.

Finally for those wishing to use the CodeTalker a User Manual can be found in Appendix D.

The full source code for all versions of the CodeTalker can be found on the accompanying CD or on /home/lscs/fyproject/CodeTalker.

2. Background

Development of voice recognition software

In the last few years voice recognition technology has improved dramatically from only being able to recognise a few specific words to recognising any voice with a wide vocabulary. This improvement is for two reasons. Firstly the software used has developed from trying to find exact matches to stored templates to finding the most likely match and from storing templates for each word in a vocabulary to storing the sounds that make up a word (the phonemes). Storing phonemes rather than words has had a large impact on vocabulary size since any word can now be broken into its constituent parts thus reducing the storage required and the system can 'guess' what a new word should sound like. The second reason is the large increase in computing power over the last few years coupled with a similar reduction in cost. This has meant that a basic level home computer is now capable of real time voice recognition.

Entry level voice recognition systems typically cost around £150, will work in real−time on an entry level PC (130Mhz processor and 32mb of RAM) and are capable of recognising around 140 words per minute. These basic packages are able to recognise discrete speech i.e. speech where the user leaves a small gap between each word. More advanced packages are now able to recognise continuous speech where the user simply talks as normal. This project has been developed using a discrete voice recognition package, since these provide better accuracy at present. The only difference between the two should be that a continuous speech system is faster to use and easier on the voice.

The only drawback with current systems is that each user still needs to spend some time training the system to recognise their voice before it will recognise them accurately. This initial training period can put people off from using speech recognition systems. As the user group being considered for this project is one that does not have the option of the keyboard or mouse they are likely to put up with this initial inconvenience in order to be able to use the computer.

The main voice recognition packages commercially available are:

Dragon Dictate

- 30,000 word vocabulary, 98% accuracy
- 140 words per minute
- Min spec: 16Mb of RAM, 45Mb Hard Disk space and 66MHz processor

L&H Voice Express

- 30,000 word vocabulary
- 30,000 word vocabulary, 95% accuracy
- 140 words per minute
- Min spec: 16Mb of RAM, 45Mb Hard Disk space and 166MHz processor with MMX

IBM Via Voice

- 140 words per minute , 95% accuracy
- 64,000 word vocabulary
- 140 words per minute
- Min spec:16Mb of RAM, 45Mb Hard Disk space and 166MHz processor with MMX
-

The general consensus among voice recognition software users is that Dragon Dictate software is the best (probably due to its higher level of accuracy). This was the software used for evaluating the project though any voice recognition software, including continuous recognition systems, will work with it.

Current uses for voice recognition software

As voice recognition has improved people have started to look at the different uses to which it can be put. One of the major uses at present is for automated telephone services, instead of using the keypad on a phone to enter details such as bank account numbers the number can be spoken. This has also opened up the type of automation possible since the company is no longer restrained to simply using numbers as they were with the phone. A good example of this kind of technology is the Wildfire assistant provided by the Orange phone network. By talking to Wildfire you can ask to be phoned with reminders, retrieve addresses and perform other voice−controlled functions.

New uses for voice technology are appearing all the time and more and more companies are starting to speech−enable their software, meaning that it can either converse with you or at least speak to you. For example Lucent Technologies has recently added advanced speech−enabled directory searching to its family of Enterprise Directory Solutions. This allows callers and phone attendants to "look up" a desired phone number, in a corporate directory of up to 1 million entries, by speaking the person’s name **[1].**

As well as providing new services voice recognition can also be used to enable people unable to use a keyboard and mouse to perform tasks that able−bodied people now take for granted, for example writing a word document and sending e−mails. Current voice recognition software has concentrated on speech−enabling these basic functions but interest groups have been quick to start up to try and enable more complex functions that do not use

standard dictation, such as programming by voice. Despite the way in which voice recognition could help such people it has not made much ground at present, a study into the disabled and computer use found that only one−quarter of people with disabilities own computers, and only one−tenth ever make use of the Internet **[2].**

Why programming by voice?

Enabling disabled people to program by voice opens the doors to a huge industry that might otherwise be closed to them. In addition it allows programmers who, due to injury, are no longer able to type to continue with their jobs. This sounds good in theory but is there anything other than a small minority who would find such software useful? The answer to this would seem to be yes.

At present there are 500,000 registered sufferers from RSI in the UK (Independent newspaper, April 23rd) , 80% of whom are habitual keyboard users. The estimated cost to industry of this is £3 billion. In general people are reluctant to report RSI, fearing that admitting to condition would affect their job. Conservative estimates estimate that only 40% of cases are reported which would put the number of suffers nearer to 1 million in the UK alone. RSI is a general term for any medical condition incurred as a result of repetitive motion. This generalisation has made many people sceptical of the term, however the main two conditions that people with RSI suffer from, tenosynovitus and carpel tunnel syndrome, are well known and debilitating illnesses not always caused by repetitive motion. With the current UK population standing at 58million this represents around 2% of the population who would find voice recognition useful. If only a quarter of these were interested in programming then that still represents a significant number.

Looking across the globe the top four fastest growing occupations in the USA are all related to computer programming with 2.8 million people estimated to be working in the 4 sectors shown below by 2008.

	Employment		Change	
	1998	2008	Number	Percentage
Computer Engineers	299	622	323	108
Computer Support Specialists	429	869	439	102
Systems Analysts	617	1,194	577	94
Database Administrators	87	155	67	77
Desktop publishing	23	44	19	73

Figure 1: Table to show top five fastest growing occupations in the USA. All figures are in thousands of jobs.

At present there are 648,000 computer programmers and over 3 million people working in the IT industry as whole [**3**]. With the USA population currently at 274 million this means that just over 1% of the population work in the IT industry.

While RSI is probably the most well−known reason for being unable to use a keyboard there are many other medical conditions that affect a person's ability to use a computer, for example tetraplegia, amputation of hands/ forearms/ arms, Stroke, birth injuries (Erb's palsy, spasticity). In 1996 1.3 million people were registered as amputees in the United States and 3,000 people that year had an upper limb amputation, if you include amputations of one or more fingers this figure rises to 26,000 [**4**]. As well as physical disability blind people may also find a voice−voice based programming system useful. Altogether, blindness and low vision limit the activities of 1.3 million people in the USA. Of the 20.9 million Americans classified as work disabled, 5 million have computers at home. Again if only a small percentage of these people were interested in programming that would open the doors to that career for thousands of people.

Another factor that increases the number of people to whom this project may be relevant is that programming is becoming a skill required in a wide range of professions, not just IT. Programming is taught as part of most scientific and engineering degree courses in the UK and the advent of macro languages for programmes like Excel has meant that people in all kinds of jobs from accountancy to personnel are now exposed to some kind of programming. There are 1.5 million trained engineers in the USA alone, not considering computer scientists.

In addition to enabling the disabled to program, programming by voice has implications for able−bodied programmers as well. The increase in mobile phone technology has meant web content and e−mail can now be received on mobile phones. One development from this is using XML to define which parts of an e−mail should be sent to the phone[**8**]. For example someone may only be interested in the sender and subject of an e−mail, if the subject looks important the full e−mail could be downloaded onto a PC. Programming by voice could allow dynamic programming of the filtering rules to apply by phoning up the computer and dictating the rule set. While the number of applications may not seem great at present the mobile phone revolution is likely to bring to light many other cases where being able to programme a computer over the phone may be useful.

Research into programming by voice

There are a few tools currently available for programming by voice but these are all the product of on−going research rather than commercially available software.

These tools have all concentrated on writing macros that manipulate the text that the user enters with their voice to make it faster and more accurate than using voice alone. In particular work has been done on enabling the Unix based editor Emacs to be used with speech As yet, no editors or macros have been developed to run specifically under a Windows environment. A version of Emacs exists that can be run under Windows so the tools should be able to be translated to a Windows environment, however all development has been done with a Unix environment in mind.

Tools currently available are:

Voice Grip [5]

VoiceGrip is an add−on software module for the Emacs editor written in Perl. It is designed to help programmers using commercial speech recognition systems to dictate source code. After code has been dictated into the editor macros can be started which run the relevant VoiceGrip code to translate the text to programming statements in any of the supported programming languages (currently C, C++ and Perl).
VoiceGrip provides support for translation of text to native code, conversion of strings to variable names, navigation of source code and searching of source code. 'Translating' in this context means recognising constructs such as loops and adding in the relevant punctuation (semi−colons, braces etc). For example:

Saying *"if current record number less−than max records then*"
Will type: `if current record number less than max records then`

After running the VoiceGrip macro this will be converted to : `if (currRecNum < maxRecs) {`

VoiceGrip is compatible with most commercial continuous speech recognition systems and runs on Win32 and Unix platforms. To run VoiceGrip, you need:

- Perl 5.005
- Emacs editor
- A speech recognition system that allows direct dictation into Emacs and invocation of Emacs macros.
- Windows 95 or NT or Solaris

VoiceGrip was developed by Alain Désilets, Institute for Information Technology, National Research Council, Canada

Java by Voice [6]

Again this is a series of macros designed to be used with the Emacs editor to allow the easy creation of Java
Java code, developed by Jonathan Epstein

Whereas VoiceGrip requires you to repeat variable, class and function names every time they are used Mr Epstein used the idea of a numbered list of variables, function and class names, defined once and then referred to by number.

Thomas Rene Nielsen's DEMACS [7]

Another package of DragonDictate and Emacs macros with special emphasis on writing source code

Programming by voice is an area still very much in the research domain and certainly not something that has been adopted by many programmers. Enough research has been done to prove that programming by voice is possible but so far no software has been designed especially for this task and no evaluations performed to find out how useful current methods are in speeding up the software development process. The VoiceCoder project is by far the most advanced in terms of reaching a commercial solution to the problem and is freely available from their web site. The latest version of VoiceGrip contains the features described above and a version of Jonathan Epstein's macros enabling names to be created and displayed in a numbered list so the programmer is free to choose the method that he finds most useful.

An alternative approach suggested by Adam Janin [**9**] was to develop a new language that was 'speech friendly'. Though this is an interesting idea it was decided that enabling people to programme by voice in existing languages was a good solution for the short term, and could allow a community of such programmers to be built up, justifying development of a specific language in the future. Thus this project focuses on enabling development in currently available languages.

Although Mr Janin discusses the features that such a language should have his research has not progressed further than this due to a waning interest as his RSI improved.

Originality of this project

The software developed within this project looks at both methods of name creation − speaking the whole name each time, and creating the name once and referring to it from there onwards as a number in a list. In addition the prototype developed at the end of the project uses context to determine when a user is dictating a function, variable or class name and formats it automatically.

The main contributions of this project to the field of programming by voice are:

- An editor specifically designed to be controlled by voice. This considers the problems associated with controlling standard mouse driven user interfaces by voice as well as simply improving the programming task.
- A full implementation of a radical new method of improving the programming by voice task that looks at the context of the text being dictated. This new method outperforms all currently existing ones.
- A self−contained package that does not require the user to have auxiliary software installed such as Perl and Emacs. The only additional software that the user needs to use the CodeTalker is a voice recognition package such as Dragon Dictate.

- A suite of software that provides a toolkit for quickly developing all kinds of voice controlled software applications, not just ones to support programming.

3. The programming task

The programming task can be broken down into 4 main areas, namely:

1. Writing
2. Debugging
3. Compilation
4. Execution

A programmer typically sits down to write a small section of an overall programme that solves a particular problem. After writing part of the code it is compiled and, if compilation is successful, run and tested. Any errors found during testing can be found and tested using debugging tools. Once the code runs correctly it is then extended to perform slightly more of the task. The process is repeated until the code solves the overall problem.

The programming task is therefore one of writing followed by correction of mistakes, testing and debugging. A programmer typically works within an environment that provides tools to support each of the four functions, for example Microsoft Visual Studio.

Time constraints mean that it is not feasible to develop a complete environment so this project focuses on a solution to the first task, that of writing the code. Although the CodeTalker does not specifically support debugging and compilation the two are still possible by voice so a programmer could perform the entire task by voice with the aid of the CodeTalker.

3.1 Writing code

Writing code typically breaks down into the following tasks (the exact ones present depends on the language being used)

- Definition of a class name
- Definition of the functions to be used
- Definition of the variables to be used within each function
- Manipulation of the variables by using other function calls or specific constructs such as for and while loops or if and switch statements.

If a specific programming task was given to a group of people each answer would look different, however simple the task. This is because every programmer has a distinctive personal style (so much so that highly accurate

programs have been developed to detect the author of a piece of code simply by the style in which it was written). There are several things that determine the style of a programmer:

- Way in which variables, class and functions names are defined
- Spacing of code
- Placing of braces
- Way in which code is indented
- Use of comments

E.g. An example of two very different styles of code that both achieve the same result is shown below.

Style A

```
public void displayCounters(int maxValue)
        {
        xCounter = 0;
        yCounter = 0;

        while (xCounter < maxValue)
                {
                        xCounter++;
                        yCounter++:
                        System.out.println(x+y);
                }
        }
```

Style B

```
Public void count(int I) {
x=0;y=0;
While (x<I) { x++; y++; System.out.println(x + y);}
}
```

Hierarchical Task Analysis

The following is an hierarchical task analysis for the programming task. It shows the different steps that the user needs to perform at each stage of writing the code. The main tasks involved in writing a programme are:

- Open File
- Edit File
- Save File
- Compile File
- Exit

Each main task is expanded below. The analysis is given for writing code in an object orientated language such as Java or C++. However the main tasks and their subtasks would be the same for any programming language used.

Grey lines indicate the level of each task within the hierarchy, level one being the highest level and six being the lowest. All tasks below level two involve some kind of typing or dictation into the editor and it the tasks below this level that the editor seeks to provide tools for.

Tasks above level two are generally thought process rather than actual actions performed, for example define new class, define main body.

Wherever possible tasks at or below level four should be automated (for example typing braces and semi−colons), since these are the most basic but often form the largest part of work to achieve each task.

Task: Open File

Select Open File option
Enter name of file to open *or*
Click on the name of the file from a list of displayed
names

Task: Edit File

Level 1 Level 2 Level 3 Level 4

Include relevant libraries
- Enter '#include'
- Enter name of library to use
- Semi-colon to finish

Define new class
- Enter 'class' keyword
- Enter class name
  - Define any class that it may extend
  - Enter 'extends' keyword
  - Define name of superclass
- Define any class that it may implement
  - Enter 'implements' keyword
  - Define name of abstract class
- Define class scope
  - Enter pair of braces
  - Position cursor inside braces
- Define any variables that the class has
  - Define variable scope (public, private, protected)
  - Define variable type
  - Define variable name
  - Semi-colon to finish
- Define constructor for the class
  - Enter keyword 'public'
  - Enter the name of the class
  - Define any parameters that the a newinstantiation of the class must take
    - Open brackets
    - Enter set of all (class, name) pairs
    - Close brackets
  - Define main body of the constructor

Define new function
- Define function scope (public, private, protected)
- Define function return type
- Define function name
- Define any parameters that the function takes
  - Open brackets
  - Enter set of all (class, name) pairs
  - Close brackets
- Define the main body of the function
  - Define variables that the function uses
  - Define any assignments to these variables
  - Print data out on screen
    - Enter 'System.out.println"
    - Enter data to display
    - Close brackets
    - Semi-colon to end
  - Use 'if ' statement
    - Enter 'if' keyword
    - Open brackets
    - Define constraint
    - Close brackets
    - Open braces
    - Define code to execute if statement istrue
    - Close braces
  - Use 'else if' statement
    - Enter 'else if' keyword

Level 1 Level 2 Level 3 Level 4 Level 5 Level 6

- Open brackets
- Define constraint
- Close brackets
- Open braces
- Define code to execute if statement is true
- Close braces
- Use ‘else’ statement

Enter ‘else’ keyword

- Open braces
- Define code to execute if statement is true
- Close braces

Use ‘for’ statement

- Enter ‘for’ keyword
- Open brackets
  - Define looping criteria
    - Define variable to use
    - Initialise to starting value
    - Semi-colon
  - Define point at which looping should cease
    - Provide comparison to an absolute value
    - Provide comparison to another variable
    - Semi-colon
  - Define amount by which variable should be incremented
- Close brackets
- Define code to execute
  - Open braces
  - Enter code to execute
  - Close braces

Use ‘while’ statement

- Enter ‘while’ keyword
- Define finishing condition
  - Open brackets
  - Define variable to use
  - Define point at which looping should cease
    - Provide comparison to an absolute value
    - Provide comparison to another variable
- Define code to execute
  - Open braces
  - Enter code to execute
  - Close braces

Use ‘do.. while’ statement

- Enter ‘do’ keyword
- Define code to execute
  - Open braces
  - Enter code to execute
  - Close braces
- Enter ‘while’ keyword
- Define finishing condition
  - Open brackets
  - Define variable to use
  - Define point at which looping should cease
    - Provide comparison to an absolute value
    - Provide comparison to another variable

Define new variable

- Define variable type
- Define variable name

Assign a variable a value

- Define variable to use
- Indicate that you want to assign it a value
- Assign a value

Level 1 Level 2 Level 3 Level 4 Level 5 Level 6

By creating a new instance
Define name of class to use
Define any parameters required
Open brackets
Define all variable names and values
Close brackets
Semi-colon
By using a function
Define function to use
Define any parameters that the function takes
Open
Define all variable names and absolute alues
Close brackets
Semi-colon
Using another variable
Define the variable to use
Define any function that should be applied to the
Using an absolute value
Define value to assign
Use function
Define name of function to use
Define any parameters that the function needs
Open brackets
Define variables or absolute values to be passed
Close brackets
Semi-colon

Save File

Select save file option

Enter name to save file under or Select save file option (if file has been previously saved)

Close File

Select Close File option

If prompted to save changes click on 'Yes' or 'No'

Close Application

Select Exit Application option

From level three downward the actions are syntax orientated i.e language dependent. It is the dictation of this syntax that is most difficult by voice and is where those programming by keyboard make the most errors. If the

syntax of the language was different, for example contained fewer brackets and allowed white space between punctuation, then programming by voice would be simpler. This raises the point that developing an entirely new language with a 'voice friendly' syntax may be a better approach [**9**]. While this idea has interesting implications developing a new language is not a simple process especially when the market for such a language is unknown. It was decided that enabling people to programme by voice in existing languages was a good solution for the short term, and could allow a community of such programmers to be built up, justifying development of a specific language.

3.2 Difficulties with programming by speech

All of the tasks associated with writing code are difficult when performed using speech. 'Difficult' in this sense means that they are either hard to do or simply tedious to the point of putting the user off from using speech.

Personal Style

Styles such as A (p15) that make use of indenting to quickly identify where braces start and end and have longer, more meaningful variable names will take longer, and be more tedious, to type by voice than more basic styles such as B. A naïve solution may be to say that programmers should simply adapt their style to make the task easier. However the increased readability of the first style is an important factor to the speed at which other people can use and adapt their code (a common task in the workplace), so any solution should attempt to fit around the user and accommodate the style that they use.

Definition of variable, class and function names

If all function, variable and class names were a single English word then this would not be problematic. In reality this is rarely the case, names need to reflect the function that is being performed or the contents of the variable, e.g. matrixMultiply or xResolution. Compilers require that these names do not contain white space so programmers have come up with various methods of concatenating words which making them easily readable. These include capitalising the first letter of each word or concatenating words with underscores e.g. matrixMultiply or matrix_multiply. The problem with this is that voice recognition software is designed for tasks such as dictation of letters and will automatically add white space between each word that is dictated. In general this is a sound design principle since one would quickly tire of saying 'space' after each word when writing a report, however it makes the dictation of these names difficult. For example, to dictate 'matrixMultiply' using Dragon Dictate requires saying:

'matrix, no space, capitalise next, multiply'. Or 'matrix, no space, underscore, no space, multiply'

Names are typically longer than just two words and the longer the name, the slower it is to dictate.

Writing constructs such as for and while.

All constructs such as loops and ‘if’ statements are generally defined by having some boolean test in brackets and then a section of code to execute if the test is true contained within a pair of braces. It is the dictation of these brackets and braces that make using these constructs laborious.

For instance to dictate the code :

```
for (count=0;count<10;count++)
{
print(count);
}
```

You would have to say: “For, no space, open brackets count equals−sign zero. Semi−colon. Count less−than ten semi−colon. Count plus−sign plus−sign .Close brackets. New line. Open braces. Print. No space. Open brackets count. Close brackets. Semi−colon. New line. Close braces”

Navigation.

This problem applies to any task that involves using speech, not just programming. The mouse has given us an excellent tool for quickly navigating around documents since it allows you to instantly move the cursor to a line by clicking on it or scroll up and down a document to find the section that you are looking for. However it is not a tool available to those that cannot use their hands. The general solution that speech recognition packages have come up with is to allow you to navigate by saying ‘move up’ or ‘move down’. You can either specify the number of lines to move or the cursor will start to move in the direction requested and you can say ‘stop’ when the it reaches the line that you wanted to go to. This works fairly well but the delay in the software recognising that you have said stop means that you often overshoot the line that you were aiming for. Slowing down the rate of scrolling can solve this but makes scrolling to the top of a document from the bottom very slow. Specifying the exact number to of lines to move means that you need to be able to count them, not easy if the line is not close to your current position.

Using the menu options

From the hierarchical task analysis it can be seen that many of the tasks require accessing menu options (e.g. opening, closing and saving files). As with navigation this is a simple task to perform with a mouse but a difficult one when using voice alone. All the main voice recognition packages use the same general method of accessing menu options which is to indicate the you want to do so (in Dragon Dictate this is done by saying ‘Command Mode’) and then giving the name of the menu and then the menu option. So to close a file you would say:

Command mode. File. Close.

Though not intuitive, this method is fine for straightforward tasks such as closing a file, but is by no means perfect. The main limitation is that menus are generally very graphically orientated and make use of check boxes, lists and dialogue boxes. While it is still possible to use lists and dialogue boxes, though tedious, changing the value of a check box seems to be impossible. This is an obvious problem as check boxes are often used to switch different functions of an application on and off.

4. Software

4.1 Introduction

Four programs were written to solve the problems of programming by voice. Three of these were used for the detailed Human−Computer Interaction (HCI) evaluation and the fourth was a prototype developed as a result of the investigation and from ideas arising during development and testing. Each program attempts to solve the problem in a slightly different way although the basis of each is the same.

The CodeTalker appears to the user as a standard word processor and supports all the features that would be expected in such an application. In addition it provides extra functionality that attempts to deal with the problems of programming by voice.

4.2 Functionality similar to all versions

The functionality described below (keyword replacement, constructor replacement, line numbering and syntax colouring) was considered essential for easing the programming task and for making it easier for the user to navigate and read the code written. Because of this it was included in each version of the editor, however menu options allow the user to turn the different functions on and off as they please.

Keyword replacement

The user is able to define so−called keywords that can be replaced by other text to speed up the programming process. For instance the user may choose to have the word ‘print’ replaced by the text ‘System.out.println();’.

A keyword can be replaced by text of any length and for greater control the user can also define where they want the cursor to be positioned after the replacement. The user is able to add to, and remove from, the list of keywords and their replacements as they wish.

The simplest method of keyword replacement is to replace the text once the user has dictated it, without prompting them. Although other methods could have been employed, for example displaying the replacement and asking the user to confirm that they wanted the text replaced, these alternatives only seemed to add to the time taken to complete the task without making it any simpler. For this reason the same method was used for keyword replacement in each editor.

The CodeTalker currently recognises .html and .java file extensions and the keyword replacements made depend on the extension that the file is saved with (i.e. they are language specific).

Constructor replacement

As well as replacing keywords the CodeTalker also recognises for, while, and do while loops and automatically adds in the formatting and punctuation required for them. For instance to dictate the code:

```
For (count =0;count<10;count++)
{
}
```

The user would say ‘for count equals zero. Count less−than ten. Count plus plus’. This makes the dictation as natural as possible and saves time by removing the need to dictate the punctuation.
Unlike the keywords the constructs are hard coded for each language that is supported and cannot be changed by the user.

The same reasoning for using the same architecture across all versions holds for constructor replacement as held for keyword replacement although there is one slight difference – version A replaces each word as it is dictated whereas versions B and C wait for the user to dictate the entire text of the constructor before replacing it all in one go.

White space removal

As Dragon Dictate and other voice recognition software has generally been designed with dictation of letters and other documents written in ‘normal’ English in mind, it automatically puts spaces into text to make it read properly. This has already been mentioned as a problem with regards to creating variable, function and class names but it also a problem when dictating brackets, semi−colons and other punctuation marks required in programming.

e.g. Saying ‘count equals−sign test dot counter open−brackets, close−brackets, semi−colon’ would type:

```
count = test. counter (  ) ;
```

This would be considered illegal by almost any compiler for any language since they require no spaces between class names and function calls and the end of a line and a semi−colon, i.e. it should read ‘count = test.counter();’

Dragon Dictate can be set to not put spaces after exclamation marks or full−stops but you cannot define how it treats brackets or other punctuation marks. Changing these settings reduces the problem but means that you need to re−set them should you want to stop programming and start to write the documentation, not an optimal solution.

White space removal attempts to solve this problem without needing to alter the settings of the voice recognition programme.

The CodeTalker keeps a list of the punctuation marks that should not have white space before them. If it sees one of these punctuation marks being dictated it checks the character preceding it in the editor and, if it is white space, removes it. The same is done for a list of punctuation marks that should not have white space after them. Although this may seem trivial it does speed up dictation greatly since every line of code ends with a ‘;’ which the user would otherwise have to say ‘no space’ before dictating.

Characters that should have white space preceding them removed:

open brackets ‘(‘, full stop ‘.’, semi−colon ‘;’, plus sign ‘+’, minus sign ‘−‘

Characters that should have white space following them removed:

semi−colon ‘;’, close brackets ‘)’, full stop ‘.’, exclamation mark ‘!’ , plus sign ‘+’, minus sign ‘−‘

Line numbering

Line numbering is provided and by saying the command ‘goto line xx’ the user can quickly jump to any line within the program.

Syntax colouring

Syntax colouring is provided and numbers, strings, comments are all coloured in different ways for easy identification. All language specific words such as for, else, public, private etc are also coloured. In addition the CodeTalker also highlights the text of commands as they are being entered. This gives a very clear visual cue to the user that they are dictating a command, not just entering text. Normal text is left coloured black.

Since syntax colouring and line numbering can really only be performed in one way, colouring the text as it is written and displaying line numbers down the side of the editor window, the implementation used was the same in each version of the editor.

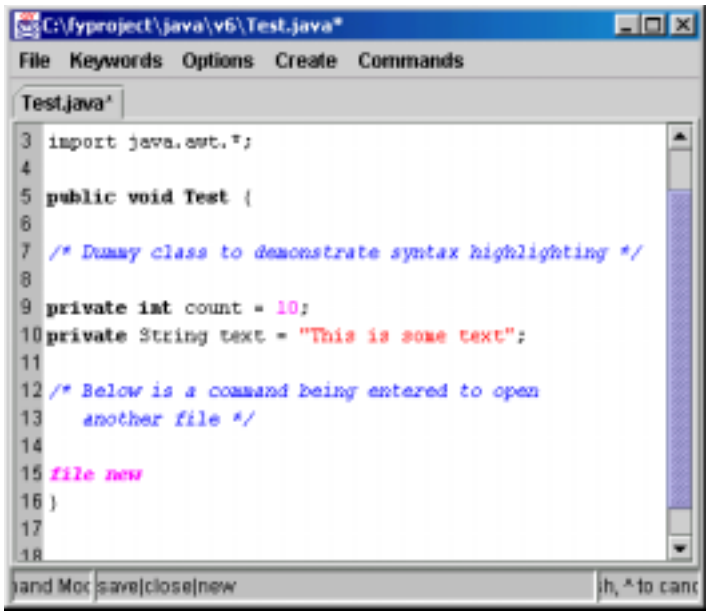

Figure 2: Screen shot to show the different types of syntax colouring applied. The 'file new' text at the bottom is a command that is being entered

4.3 Speech Directed Interface

Although the HCI evaluation and this report concentrates on the ways in which the programming task has been made easier the word processor itself has also been designed to be as speech−friendly as possible.

Nearly every software package developed nowadays has a menu bar along the top of the screen which, when clicked on, display different options to the user. As mentioned before this is a good method if you have access to a mouse but is difficult to use by speech alone.

The word processor has several features designed to overcome these problems (all features are included in each version)

- On−line command parsing
- Command prompts along the bottom bar
- Tabbed panes

On−line command parsing

Since accessing standard windows menus is not straightforward using the voice, all versions of the editor provide an alternative method of accessing the different menu option. The CodeTalker has a list of all the commands supported, for example file open, file close, exit program etc. When the first word of a command is dictated into the editor window the CodeTalker highlights the text and waits for the user to dictate the rest of the command.

Once the complete command has been entered the text of the command is removed from the editor window and the action requested performed.

At any point the user can cancel the command at which point the CodeTalker deletes the text of the command and returns to the normal text mode. As a prompt for the user the complete command is displayed to them at the bottom of the screen as soon as they start to dictate it. Filenames can be given as consecutive words which are concatenated to form a legal name. This removes the need to access any of the menus in the standard way.

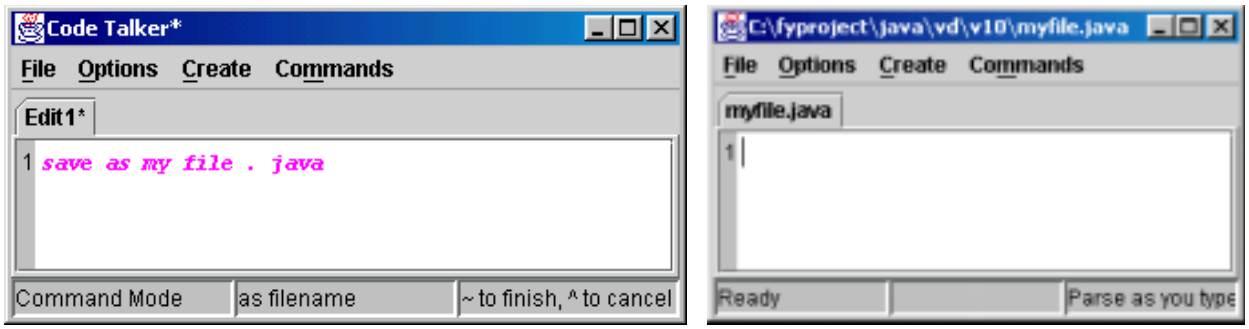

Figure 3: Invoking the 'Save as' command. Note the prompts along the bottom of the window, the colour of the text indicating that a command is being entered and the final name on the tabbed pane

In the case where the user wants to dictate words that would normally be interpreted by the CodeTalker as a command the CodeTalker can be set to wait for a specific word or character to be seen before it starts to parse for a command. The default word is 'command' although the user can change this. The 'parse as you type' text in the bottom right hand corner of the frame indicates that CodeTalker is set to parse commands as soon as they are seen rather than waiting for a keyword.

In the same way as constructor replacement was implemented, all the versions use the same base architecture but version A parses each word as it is entered whereas the other versions wait for the complete command to be entered before it is parsed and checked for errors.

Tabbed panes

The use of tabbed panes may seem accidental but is entirely deliberate. Typically a programmer will have more than one document open. The standard method of viewing the files that you have open requires clicking on the 'Window' menu which displays a list of open files. Having tabbed panes means that the names of the open files are instantly visible.

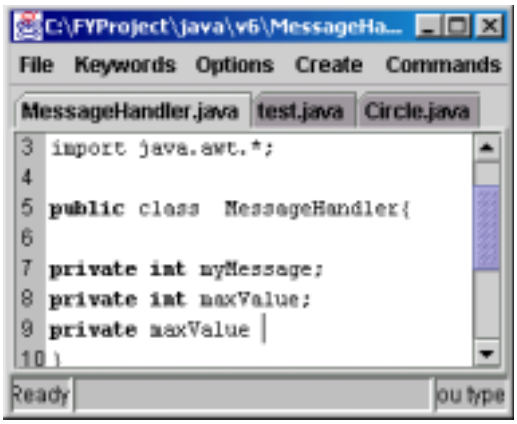

Figure 4: Screen shot showing several different files open, all easily visible via the tabbed panes

Users can move to other windows by saying 'window' followed by either by the name of the file or it's position in the row of panes. The CodeTalker will then switch the focus to the requested file.

The layout of the commands was carefully chosen to make them intuitive to someone used to accessing window−type menus either by voice or by mouse. A cognitive walkthrough was carried out to identify possible problem areas and remove them as far as possible. The results of this evaluation are given in appendix E.

4.4 Descriptions of the different software versions

Versions A, B and C differ only in the way in which names are created and the way in which commands and constructors are parsed. Version D provides a radically different way of creating names.

In all other respects they are the same and contain all the functionality described in Section 4.2.

All four versions treat variable, class and function names as different objects. While it may seem obvious to group them together as one thing, a name, this is not a good idea. Programmers generally have different ways of formatting the three so that they can easily see whether a name refers to a class, function or variable. Although the default settings mean that variable and function names are created in the same way separating the three means that a user is able to apply different formatting rules to each name type.

Version A

The user is able to create variable, class and function names, which are then listed in a series of boxes alongside the main word processing window. A name is created by saying 'create variable/classname/function' followed by the words that make up the name, terminating the command with a ~ (this terminating character can be defined by the user). On seeing the terminating character the CodeTalker removes the text of the command from the editor

window, creates the name using the method defined by the user (e.g. capitalise the first letter of each word and concatenate together) and displays the name in the relevant list box.

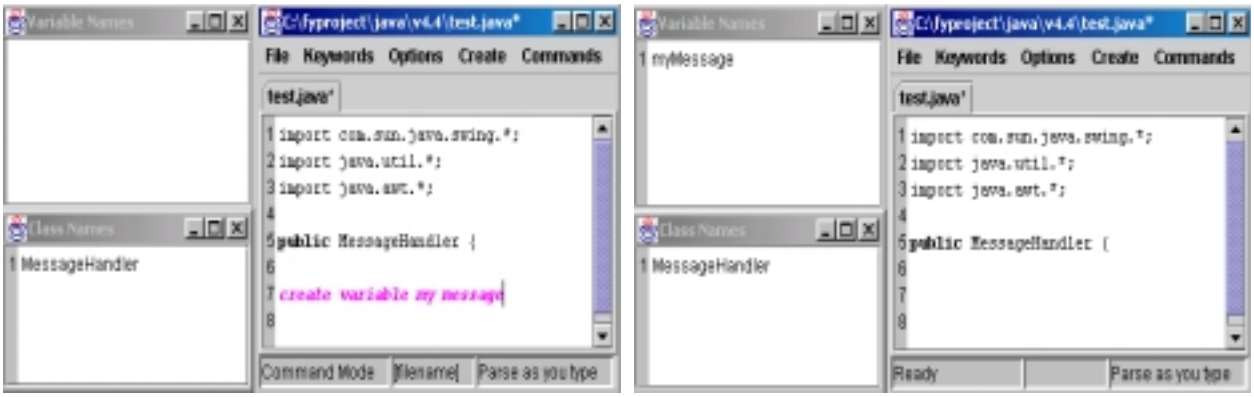

Figure 5: Screen shot showing a variable name being created. Note the highlighting of the command text and the addition of the name created to the 'Variable Names' list box.

After creation the user is able to 'type' the name by saying 'variable/classname/function' followed by the position of the name in the list box. The text of their command is instantly replaced by the name to which they have referred. E.g. Saying 'variable 1' in the editor window below would result in the text 'displayMessage' being displayed.

To make retrieval as simple as possible the list boxes are numbered and may be re−sized and re−positioned.

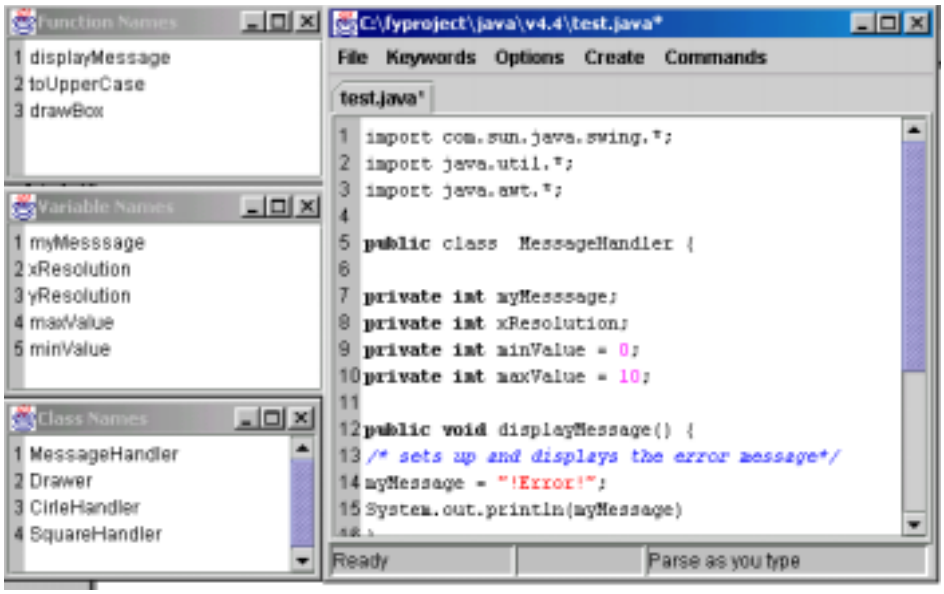

Figure 6: Screen shot of Version A showing the function, variable and class lists alongside the main editor window

When the CodeTalker recognises that a command or constructor is being dictated each word is parsed as soon as it is spoken and any errors immediately flagged.

Version B

The method of creating names is exactly the same as for Version A, however when using names the reference to the name needs to be completed with a terminating ~ before it is replaced by the name from the list.

The only other difference between the two versions is that Version B waits for the user to say a terminating character (~) before it parses a command phrase. At first glance Version B may seem redundant since it requires more words to be spoken by the user before the command is carried out however it provides the user with a greater degree of control and is more robust to errors in dictation. The level of errors and degree of control required depends entirely on the user.

Version C

The method used to create names in version C is different to that in versions A and B. Instead of using a numbered list a name is created by saying ‘create variable/classname/function’ followed by the words that make up the name, terminating the command with a ~. On seeing the terminating character the CodeTalker removes the text of the command from the editor window, creates the name and displays it in the editor window. For example saying ‘variable my message tilda’ would result in ‘myMessage’ being displayed in the editor.

The word ‘classname’ rather than ‘class’ is used to indicate that a class name is being dictated since ‘class’ is a java keyword and needed for writing Java code.

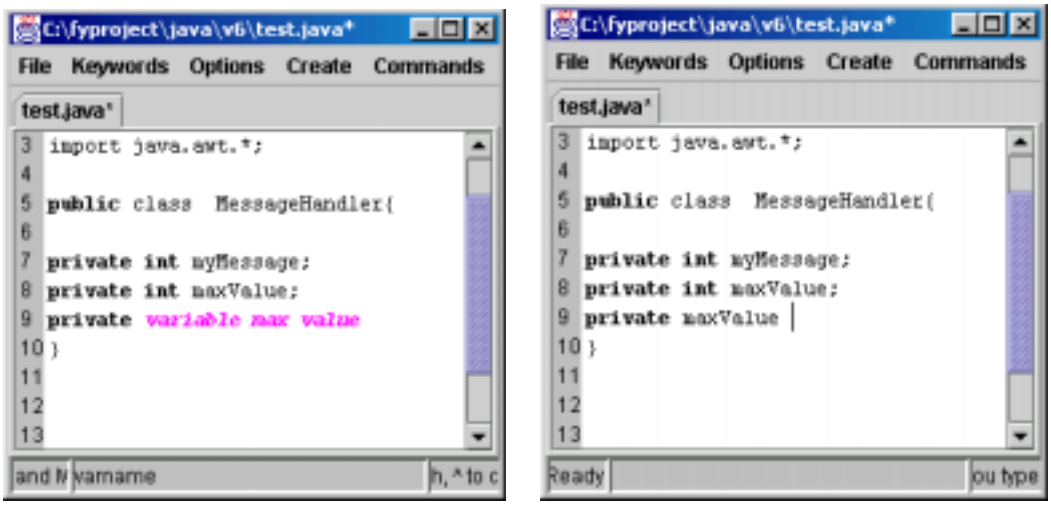

Figure 7: Two screen shots of Version C showing a variable name being created

The user needs to dictate the full name every time it is referred to (as opposed to Versions A and B where, after creation, the name can be referred to as a number).
In all other respects version C is the same as version B, i.e. the CodeTalker waits for a terminator before parsing commands and constructors.

Version D

This was a prototype version developed as a result of an idea that I had during the HCI evaluation of the other versions. It provides a radically different method of creating names that is more intuitive than the others described.

It is based on the idea that, in any language, there are certain language specific words that precede a variable name, a class name or a function name and other language specific words that indicate the end of such a name. By looking at the context in which the text is being spoken the CodeTalker is able to decide whether general text or a name is being spoken and what type the name is. This is made much simpler than it sounds by observing that if a word is not a language keyword it must either be a string, a number, punctuation or a name. Detecting the difference between these is very easy. After determining when the name has been completely dictated the words that make it up are concatenated in the usual way. The user does not need to say anything to indicate that a name is being spoken or say a terminator to indicate that they have finished.

For example, in Java any word following the word ‘class’ is a class name. As soon as the CodeTalker sees a ‘{‘ or ‘(‘ it will assume that the words between the word ‘class’ and the bracket is a class name and will concatenate the words to form one.

The CodeTalker uses a traffic light system to indicate to the user what it is doing. Normal text is written in black, words that it thinks form a variable name are written in green until the name is complete in which case it is replaced by normal text, function and class names are similarly displayed in amber and red respectively.

In addition the user is still able to say the words ‘classname’, ‘function’ and ‘variable’ to indicate that they want to dictate a specific name and can tell the CodeTalker to stop and concatenate a name at any point by saying the terminating character. This provides a greater deal of control for the user and solves the problems involved in writing sentences such as

```
MyClassName thisVariable = new MyClassName();
```

Here the CodeTalker would not know which of the words ‘my class name this variable’ to concatenate to form the class name and which to concatenate to form the variable name. Explicit use of the terminator is able to overcome this.

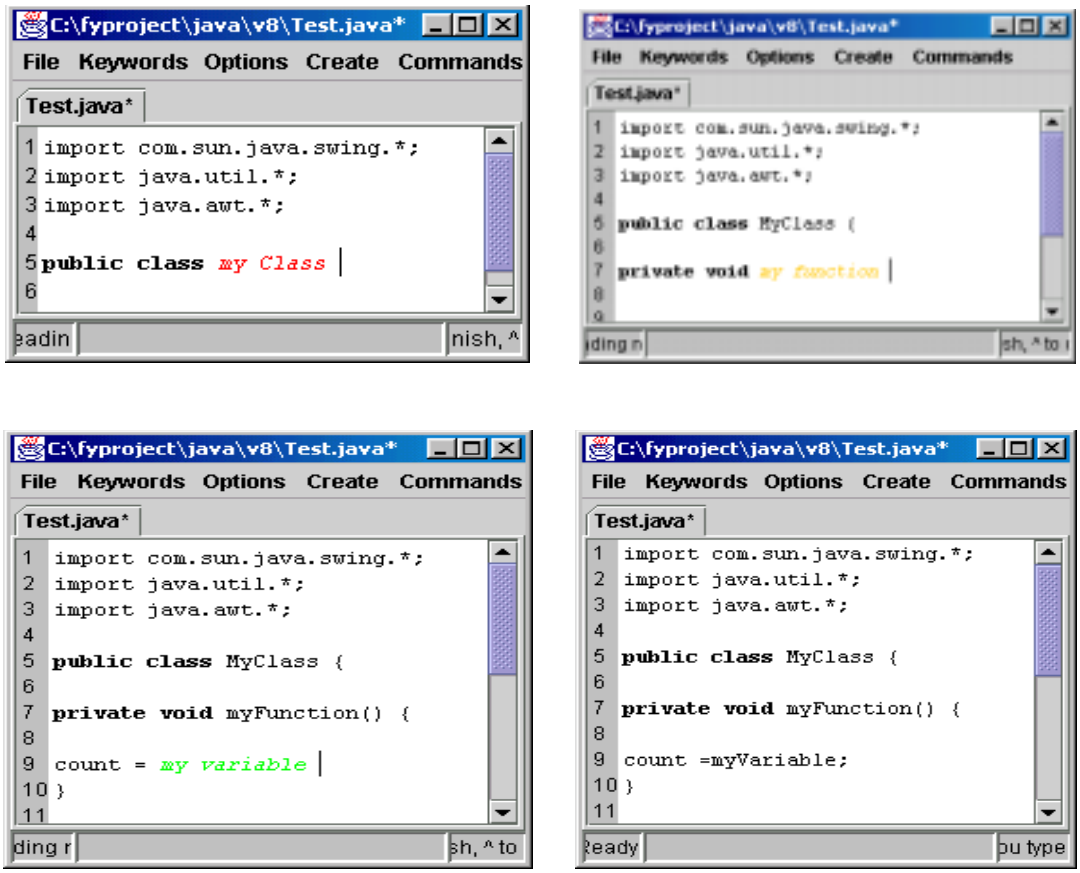

Figure 8: Screen shots showing a class, function and variable name being automatically recognised. The only words spoken were 'public class my class, braces, private void my function, open brackets, close brackets, braces, count equals my variable, semi−colon'.

4.5 Choice of development language

The software developed has all been written in Java. There were several reasons for this choice of language:

- Low level access required
- Platform
- Use of the code as a toolkit for voice orientated software

The software developed essentially captures the text being sent by the voice recognition software and 'massages' it into computer code, which is then displayed to the user in a text editor. The software needs to be able to add and remove to the text displayed in the editor as well as apply formatting to it. This requires a low−level access to the data being sent to the word processor. Development originally started using Visual C++ since that allowed rapid development of user interfaces, however it proved extremely difficult to gain this low level access required to the text being displayed in the editor. Java provided a far simpler interface to this data and no headway was really

made until switching to this language. Perl provides similar low−level access but cannot be compiled into a stand−alone executable as Java can.

As developers all work on different platforms any programme developed needs to work on all the major platforms, Windows, Linux and Unix. Java is fully compatible on any of these platforms.

As well as needing a language that provided the right functionality part of the project is the fact that the software developed can be used as a toolkit for creating other voice driven programmes, e.g. a drawing package. For a toolkit to be useful to other developers it needs to be in a language that they understand and that is well supported. The most common languages used for developing at present are Java and C++ so it would seem sensible to develop in one of these. Of the 2 Java is the more cross−platform compliant. In addition there is only one version of Java, that developed by Sun, and a central repository provides comprehensive documentation, packages and compilers. In contract C++ is far more fragmented with different companies offering slightly different versions (Borland, Microsoft etc all sell C++ compilers).

For these reasons Java was the language chosen for development.

The central repository for Java is at http://www.java.sun.com. Documentation and the latest version of Java can be downloaded from here for free. In addition it provides free tutorials and discussion boards for any problems that you might have.

4.6. Architecture

How the software interacts with voice recognition systems

The CodeTalker essentially creates a buffer between the text being sent from the voice recognition software and the text editor in which it is displayed.

The voice recognition software has no knowledge of this buffer, it merely sends the text that the user has spoken to what it perceives as a text editor. The CodeTalker intercepts this text before it is displayed and parses it. If no action is required on the part of the CodeTalker the text is simply displayed as it was spoken otherwise the appropriate action is performed and only the relevant text is displayed.

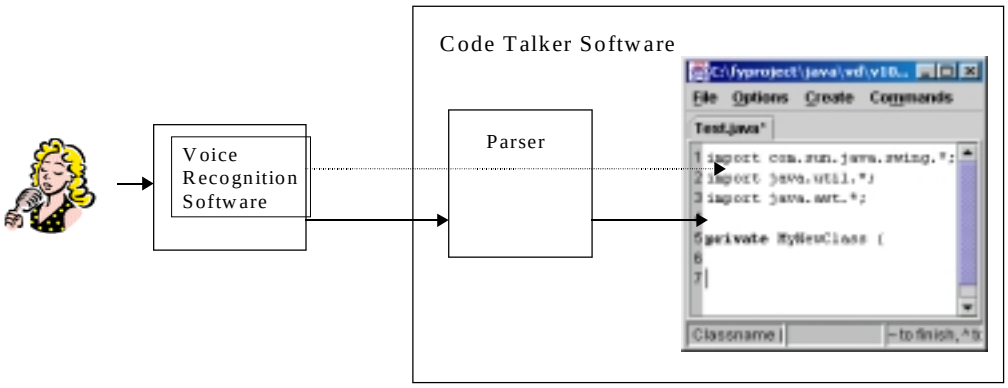

Figure 9: Diagram showing how the voice recognition software interacts with the CodeTalker. The dotted line indicates the interaction as perceived by the voice recognition software

The invisibility of this buffer to the voice recognition software means that, although the project was developed in conjunction with Dragon Dictate the CodeTalker will work with all currently available voice recognition software.

State charts

The high level state chart of the CodeTalker is the same as that for a standard text editor.

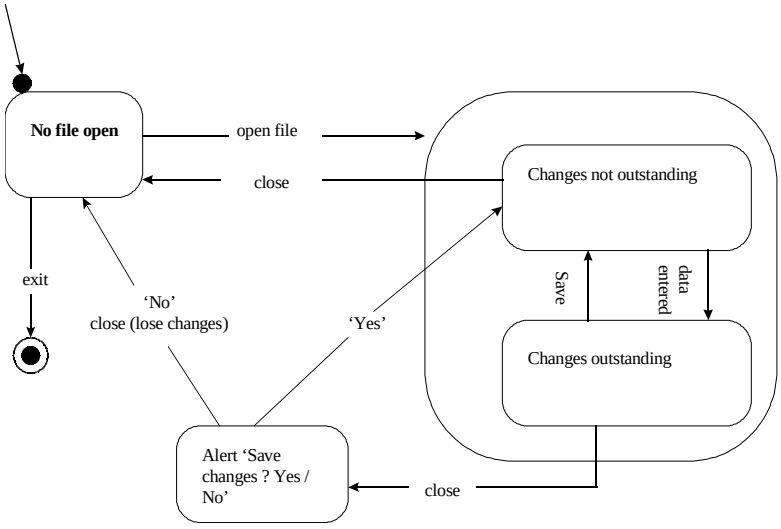

Figure 10: High level state chart for the CodeTalker program

To start with no files are open. As soon as a file is opened, the CodeTalker enters a new state which tracks whether changes have been made to the file by registering when the voice recognition software passes words to it. If a user tries to close a file that has outstanding changes they are alerted and asked whether they would like to save the changes.

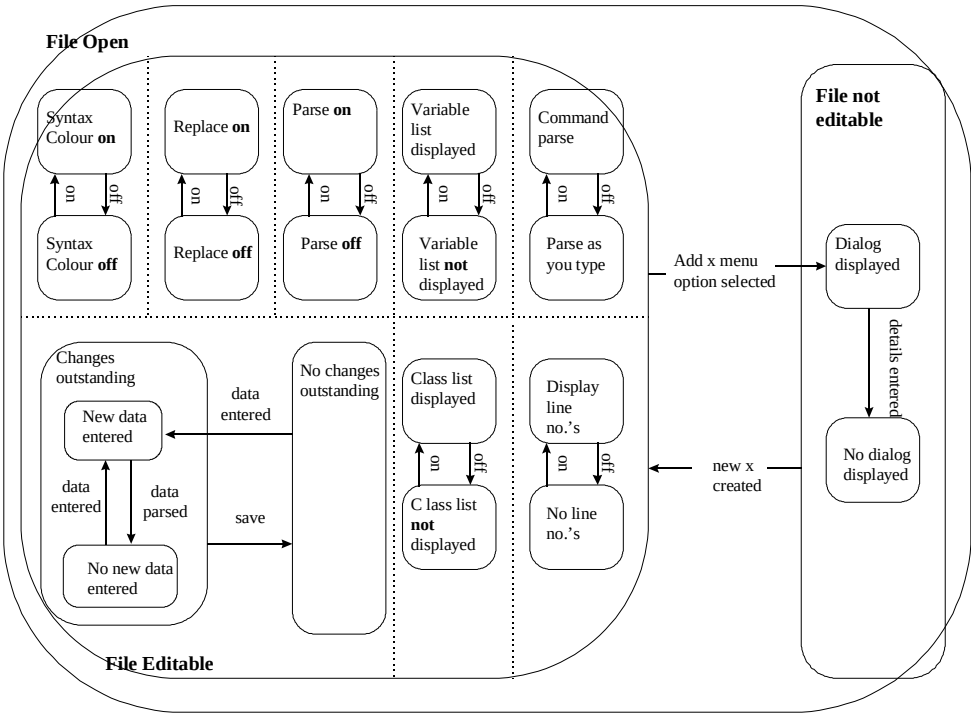

Figure 11: Detailed state chart for the 'File Editable' state

If the user dictates a command that requires a dialogue box to be displayed, e.g. asking to create a new replacement, the state of the editor changes from editable to not editable, a dialogue box is displayed and data captured. Once the user has finished dictation into the dialogue box the command requested is performed, the dialogue box removed and the editor state returned to editable.

The main algorithm with the CodeTalker tracks the new data entered and decides what to do with, whether to highlight it, replace it with other text, parse it as a command, apply some kind of syntax colouring to it or simply display it as spoken. Figure 12 shows how the different states involved with this main algorithm interact.

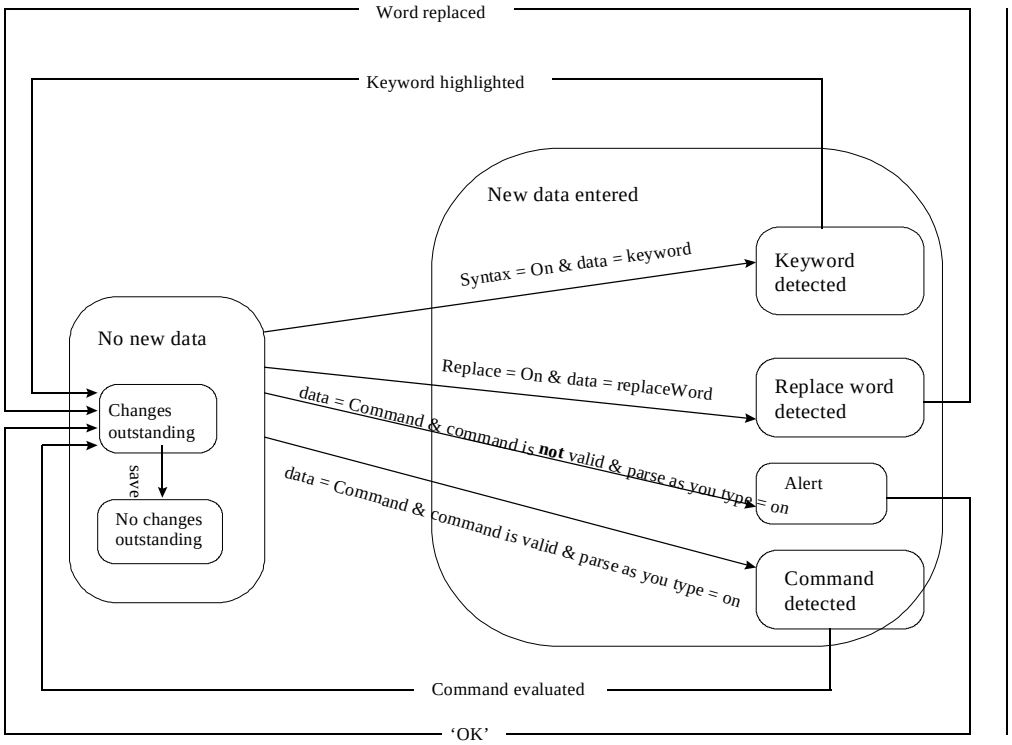

Figure 12: Detailed state chart for the 'New Data Entered' and 'No New Data' states

The Main Classes

The key classes that control the CodeTalker are the CodeTalker class, the CodeDocument class, the DataManager class, the Command class and the Parser class.

Auxiliary classes extend the Parser class to define how to deal with explicit language constructs and additional classes exist to define the behaviour required when different commands are invoked.

CodeTalker

The main class for the programme. This sets up the different panes and windows within the CodeTalker (the menu bar, different editor windows and tabbed panes). It provides a portal through which other classes can access these windows without knowing anything about their underlying implementation.

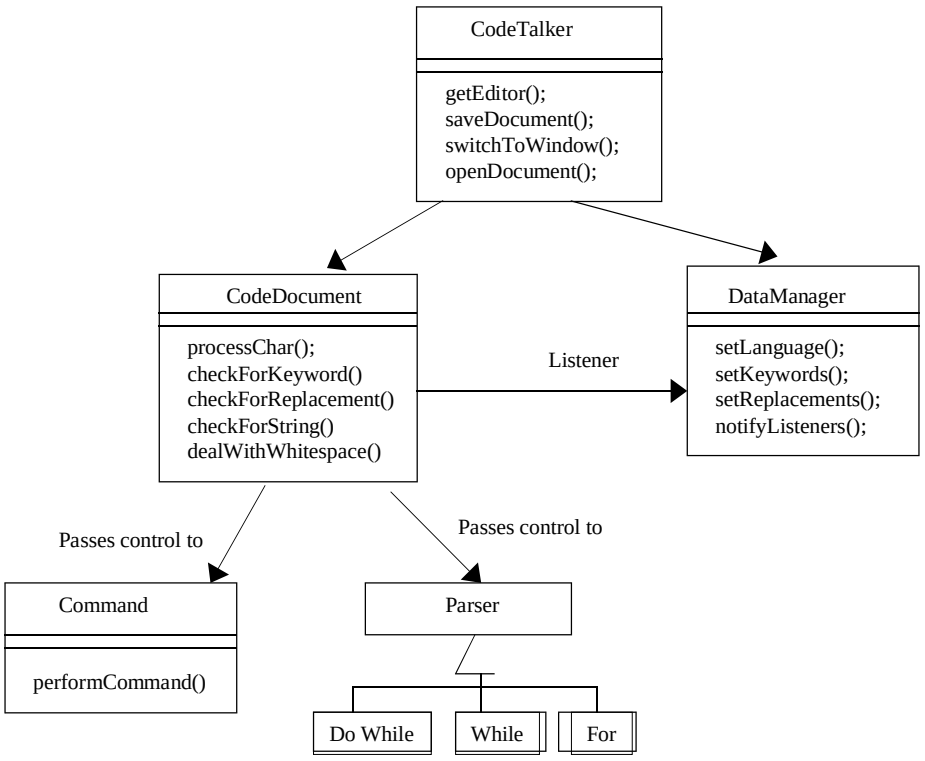

Figure 13: Class Diagram for the CodeTalker showing the main classes and functions

CodeDocument

Controls the visual display of an actual document. Each window that is opened has its own instance of the CodeDocument class. It is this class that performs the buffering between the text displayed to the user and the input from the voice recognition software. As text is received from the voice recognition software the CodeDocument class analyses what to do with it. Any commands are passed to the Command class for parsing and any constructs are passed to the Parser class. The CodeDocument class itself performs all other functionality such as syntax colouring, keyword replacement and white space removal.

DataManager

This class handles the different file extensions that are supported. CodeDocuments are able to register as listeners to the DataManager which will notify them of the words that they should be replacing or highlighting and of any changes to these words. Whenever a file is opened or saved the name of the file is passed to the DataManager which checks that the valid information for the file extension is loaded. This means that several files of different types can be open at the same time and still be correctly displayed, whenever the document being displayed changes the DataManager checks that the correct keywords and syntax to highlight are loaded.

Command

This class deals with commands being entered. It is able to parse the command and check that it is valid, providing a warning to the user if there has been a mistake. It converts any numbers or comparisons entered from words to figures. Once the command has been completely and correctly entered the details entered are passed to the PerformCommand class. This class either performs the command itself or calls the relevant class depending on the data entered.

When passed control by the CodeDocument the Command class is given the structure of the command as a vector. Each element in the vector represents the next word of the command, if there is more than one option then the options are stored as a '|' delimited list. Another vector, containing the actual details of the command entered is built up as each word of the command is spoken.

For example the command to turn keyword replacement on and off is 'replace on' or 'replace off'. The command structure for this will be:

CommandStructure[0] = ("replace");
CommandStructure[1] = ("on|off");

The vector containing the actual command entered may be one of two:

CommandEntered[0] = ("replace");
CommandEntered[1] = ("on"); or CommandEntered[1] = ("off");

The Command class tracks the data being entered but will wait for a white space, signifying the end of a word, before performing any parsing. At this point this word entered is read in and compared to see if it is valid. This overcomes the problem of a word being dictated that contains a command word within it, e.g. '**file**name'.

Why not notify the user as soon as there is an error ?

Consider the command 'replace on' which turns keyword replacement on. If the user says 'yes' instead of 'on' when invoking the replace command the parser will know that the word is incorrect after seeing the first letter but will wait until the whole word has been entered before alerting the user. This may seem clumsy but the reason is that using voice recognition software it is not like having a user typing who will stop as soon as they see an error. Once the voice recognition system has recognised the word the user has said it will send each letter to the CodeTalker, regardless of whether the CodeTalker brings up dialogue boxes saying that it is wrong. If an error were reported before the end of the word saying 'yes' would display up to three error dialogue boxes, one for each

letter that was incorrect. This is obviously frustrating to the user since they think that they have only made one mistake – saying ‘yes’ instead of ‘on’.

Displaying an error message only the first time an error is encountered would be a solution but it is not optimal. Every time a letter is received by the parser, either the sub−string received so far or the letter received needs to be checked against the expected text. This requires keeping track of the next letter expected or manipulating the text to compare the substrings for every letter of the command. However the second method only requires the CodeTalker to check if the letter received was a space. If so, the complete word is retrieved and compared against the stored template. This is computationally far less expensive.

Parser

This class deals with the parsing of constructs such as for, while and do while loops. It provides a base class that is able to recognise numbers, variables, comparisons, updates (such as ++,−−,+− etc) and exact matches. If the data dictated differs from that expected it handles all error messages and is able to delete text or interpret it at any point. By extending this class new constructs can be handled by defining their layout and the punctuation that needs to be added.

4.7 Key problems when interacting with voice recognition systems

- Voice recognition software is not 100% accurate. Any software developed needs to be robust enough to deal with this.
- Once the software has recognised a word it will send the complete text of that word to the application **regardless** of what the application does. I.e. any error messages will be ignored until the complete text has been sent. This needs to be taken into consideration when performing error handling.
- It is important to test any software developed with the voice recognition software at all times rather than emulating the it by typing. This is because voice software often has odd features that will not come to light unless tested. One example is that, with Dragon Dictate, instead of dictating a word with a space at the end it dictates a space and then the word. This may sound like a small difference but means that it is very difficult to detect when a word has been completely dictated.

4.8 The software as a toolbox

Although the software written has been developed with the idea of supporting programming by voice in mind it could easily by extended to support other voice directed activities.

The main algorithm of the software involves tracking the text entered by the user and responding in a predefined way if certain key words or phrases are spoken. In the case of the CodeTalker the software invokes functions to replace text with other text or perform standard word processing tasks such as opening or closing files. However

using the framework provided by the software different classes could be added that responded to key words or phrases in different ways to create other voice driven software.

For example:

- A drawing package could be written by replacing phrases such as ‘circle size 10 centre 15 10’ by the specified shape at the given co−ordinates
- An e−mail system that responds to the command ‘send to lscs @ doc . ic . ac .uk’ by sending the text in the window as an e−mail to the given address.

The CodeTalker has been written in a way such that all the commands and keywords recognised are controlled by just a few classes and very few changes need to be made to change or add to its functionality. For example it can be extended to recognise new commands, as described above, by adding just three lines of code and a class to perform the command function. In this respect the software can be viewed as a toolkit for developing reliable and user−friendly voice based systems.

Using the toolkit the author was able to develop a simple voice−controlled drawing package in less than three hours.

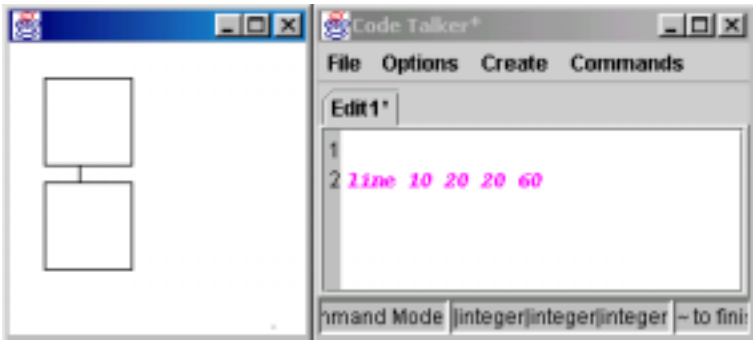

Figure 14: A simple voice−controlled drawing package developed using the CodeTalker toolkit

The way in which functionality can be added or adapted is described below.

4.9 Adding new functionality to the CodeTalker

Adding new functionality requires the user to change, or add to, the source code for the CodeTalker. This assumes that the user is able to write Java but since the user is by definition, a programmer, this is a fair assumption.

Providing an automated means of adding new functionality would require restricting what the user can add to what you think they will need to be able to do. This is very restrictive since there is no way of predicting what features

will be needed in the future, allowing the user to directly write the code gives the user far more versatility and does not restrict what they will be able to do.

Extending the CodeTalker to work with additional languages

The CodeTalker currently supports programming in Java and HTML but is written in such a way that adding support for other languages is very simple, only requiring one line of code to be added.

After a file is saved the CodeTalker calls the DataManager class which looks to see if it recognises the file extension. If it does it opens the relevant *.case, *.repl and *.keyword files and loads the data that they contain into two hashtables and a vector respectively. These files contain details of the words to replace and the language−specific words that should be highlighted. This information is then passed to the CodeDoc class which performs the conversion of data from the voice recognition software to code to be displayed to the user.

To add a new language

1. Associate the file extension with the new language. Do this by adding the new file extension and language name to the language hashtable in the DataManager class.

   For example to associate *.foo with the language 'bar' add the line
   languages.put(".bar","foo");

2. Define the keyword replacements to make and the words for the language that should be highlighted.

   There are three ways to do this:
   - Save the file with the relevant file extension.
     - Using a mouse click on the 'New Keyword' or 'New Replacement' from the 'Create' menu and type in the details *or*
     - Say 'create keyword' or 'create replacement' to bring up the create dialogue and then say the details
   - Edit the files directly.
     There are three files:
     - a *.keyword file holds each word that should be highlighted separated by a '|'.
     - a *.case file holds single words that should be replaced by the same word but capitalised, e.g. string and String. Each pair of words is separated by a '|'
     - a *.repl contains a keyword to replace followed by the text to replace by, again all separated by a '|'.

The CodeTalker will now recognise the new file extension and perform any replacements or text highlighting that have been defined.

Adding new constructs

If the new language contains no language constructs (for, while or do while loops) only the new keywords and replacements need to be defined. However if the language contains specific constructs you want to automate then it is necessary to:

1. Extend the Parser class to handle the new constructor
2. Add a line of code to the CodeDocument class to specifying the name of the new class and associating it with the specific language name.

Constructs are handled by a parser class which is able to recognise numbers, variables, comparisons, equal signs, updates (++, −−, +=, −=) and also look for exact matches. To define how to handle a construct a class needs to be created that extends the Parser class and defines the following six things:

1. The name of the constructor (for, while)
2. The starting punctuation to insert
3. The punctuation to insert after each part of the command
4. The final punctuation to add
5. The format of the constructor
6. The final position of the caret after the constructor has been parsed

The format of the command is stored as a vector of vectors called command. For example the following class defines how to handle a constructor called foo:

```
public Foo (VoiceCoder v, CodeDocument d, StatusBar s) extends Parser {

super (v,d,s);
startingPunctuation = "(";
finalPunctuation = ")\n{\n\n}";
finalCaretPosition = 2;
parsingName = "foo";

 cOne.addElement("variable");
 cOne.addElement("number");
 cOne.addElement("string");

punctuation.addElement(";");

cTwo.addElement(“increment”);

command.addElement(cOne);
```

```
command.addElement(cTwo);
  }
```

This code creates a class that will look for the word ‘foo’. On seeing it the punctuation defined by startingPunctuation will be inserted into the editor and the CodeTalker will wait for the user to say a variable, a number or a string. After that it will write the punctuation ‘;’ and then wait for the user to enter an increment operator before writing the punctuation defined by finalPunctuation to the editor and then returning control to the CodeDocument class. The final caret position will be two spaces back from the end of the text.

The parser is able to recognise the following phrases:

Word added to the command vector	Matched to
variable	Maps to a variable definition, e.g. ‘variable one’ in CodeTalker version A and B or ‘variable my Message’ in version C
number	Matches to any number
comparison	Matches to both punctuation: >, <, >=, <=, != or words: greater than, less than, greater equals, less equals, not equals.
equals	Matches to the punctuation, = , or the word equals
update	Matches to both punctuation, ++,+=, −−, −= or words: plus plus, plus equals, minus minus or minus equals.
string	Matches to any single words spoken

If any other words are added to the command the parser will try to match it exactly.

The Parser super class will automatically deal with any errors, display the constructor structure to the users when they start to speak and performing all the replacements and insertions of text.

Extending the CodeTalker to handle new commands

All of the menu options on the CodeTalker can be accessed by spoken commands e.g. file save, file close as well as other commands such as create function. In the future new commands may be needed either to control additional menu options or to provide extra functionality. The modular design of the CodeTalker means that this is a straightforward task.

The classes involved here are PerformCommand and Editor.

The Editor defines the set of commands that are recognised. Every time the user speaks the CodeDocument class looks at the word spoken and checks to see if it is a command, having been given the list of commands to look for by the editor class. If it is a command the text is highlighted to give the user a visual cue that a command is being spoken and the CodeDocument passes control to the Command class. This reads in the command as it is spoken and, once the complete command has been entered, passes the command to the PerformCommand class. This either handles the command directly or passes it to the relevant class.

To add a new command three things need to be done:

1. Define the class that will perform whatever functionality you want to occur when the command is called or add a function to the PerformCommand class.

2. Add the command structure to the command hashtable in the Editor class.

   A command is specified by the first word to be spoken followed by the rest of the command, where there are different options that can be said these are separated by a '|'.
   For example to define the command for turning syntax highlighting on and off:
   commands.put("syntax","highlighting on|off");

3. Add an 'if' statement to the performCommand function in the PerformCommand class that calls the class or function that you defined in step 1.

To remove a command just the single line adding the command to the Command hashtable in the Editor class needs to be commented out.

5. Human−Computer Interaction Evaluation

The idea of the HCI evaluation is to determine whether the features of the CodeTalker improve the programming task. The features designed to help with programming are:

- keyword replacement
- white space removal
- constructor replacement
- variable, class and function name creation

As well as making the programming task more efficient by speeding it up the CodeTalker should provide usability by making the task effective and satisfying. Dictating code using a standard text editor is not massively slow but is so tedious that it is unlikely that a user would continue to programme via this method for any length of time. The users perception of the CoderTalker is therefore as important as the actual efficiency increases themselves since a system is only useful if people are willing to use it.

The different features are evaluated in two ways. Firstly by a standard keystroke analysis to determine whether the number of words required to achieve a task is reduced by using the CodeTalker as opposed to a standard text editor. Secondly by a user evaluation to determine actual increases in speed and how easy or useful the user perceives the functions to be, again compared to a standard text editor. Before any analysis was undertaken a cognitive walthrough of the different tasks identified by the hierarchical task analysis was undertaken to identify and rectify any potential problem areas, e.g. commands that used different syntax to that which a user would expect. The final results of this evaluation are given in appendix E.

All analysis was done using the CodeTalker to programme in the Java language. An assumption made here is that as all programming languages contain similar structures and variable, class and function naming conventions the results for programming in Java can be considered to hold with any language for which the CodeTalker may be configured.

For both evaluations a set of small test programs was developed. These were designed to test all the functionality of the CodeTalker while being short enough to make the HCI evaluation possible in the given time frame. A full listing of the three test programs is given in Appendix A. The programs have been carefully designed to reflect the distribution of variables, functions and constructors that would be found in a typical program. Each program contains the same number and type of functions and variables and the same number of replaceable keywords. Each program contains one of three Java constructors (for, while or do while loops).

The standard text editor used for all evaluation was UltraEdit. This provides a straightforward editor as well as syntax colouring and line numbering. It was chosen because it is a standard editor used by programmers and it is likely that it would be used with voice programming if no specifically designed alternative was available.

5.1 Keystroke Analysis

Unlike a standard keystroke analysis where it is the number of keystrokes and mouse movements that are measured, here the number of words that need to be spoken are considered.

The number of words required to write each of the three programs using the different versions of CodeTalker and a standard text editor was measured. For each version the words required when using keyword replacement, constructor replacement and white space removal on their own and combined was measured. This allowed the relative benefits of each version as well as the benefits of keyword replacement, constructor replacement and white space removal to be measured (the different versions providing different methods of creating variable, class and function names). For each scenario (version type, keyword replacement on / off etc) the % improvement in the number of words that need to be dictated compared to using a standard text editor was measured. Since people dictate text at different speeds the keystroke analysis does not attempt to estimate the time that the task will take merely the improvement that should be found when using the CodeTalker compared to a standard editor.

The actual number of distinct words contained in each program was also measured to give a benchmark for the number of words that would be required if you had a perfect programming by voice package. A distinct word being a word that is surrounded by white space e.g. ‘myMessage’ or ‘print’.

Assumptions made

1. Variable and function names are written by concatenating the words and capitalising the first letter of each word after the first e.g. myMessage, maxValue
2. Class names have the first letter of every word capitalised, e.g. CountDown, CountUp
3. No formatting of the text using the tab key is performed.
4. Every line is completed by a carriage return.
5. All phrases that do not require pauses are counted as one word and each word requires the same time to dictate.

All evaluations were made using Dragon Dictate which provides the commands ‘No space’, ‘Capitalise Next’ and ‘New line’ for formatting text. Dragon Dictate requires a small pause between each word that is spoken, this is a similar feature for most speech recognition systems as continuous speech systems have only just started to come on to the market. A command that has to be spoken continuously to be recognised such as ‘No space’ is counted as one word as it only takes the same time to say as any other single word.

Limitations

White space removal removes white space from around all punctuation marks that would result in an invalid line of code were it to be left in, e.g. ‘this. myFunction () ;’ is invalid whereas ‘this.myFunction();’ is not. However Dragon Dictate realised that white space was being removed from before a closing bracket or a semi−colon and adjusted itself to do it automatically. Once this happened there was no way of requesting it to stop, so this white space removal was performed during the users evaluation even when they had white space removal off. This in itself was interesting behaviour since it meant that the white space removal feature may be useful for training the voice recognition software but then ultimately redundant. However Dragon Dicatate did not pick up on the fact that opening brackets, plus signs and exclamation marks also needed to have white space removed. When the users dictated code with and without white space removal the improvements could therefore only be seen in lines of code that contained opening brackets, exclamation marks and plus or minus signs.

The keystroke analysis aims to give an indication of the expected improvements which can then be compared against the actual improvement to gauge how effective the method is and how easy it is for the user to use. If the keystroke analysis were done for all white space removal this would skew the results and make the white space removal method used look very inefficient. To get a realistic idea of the actual efficiency the keystroke analysis only considered the white space that the CodeTalker was still left to remove, i.e. white space before opening brackets, exclamation marks and plus or minus signs. This means that the actual improvements between using a basic editor and the CodeTalker may be greater with a different voice recognition package that inserts white space after all characters.

Results

Figure 15: Graph to show improvement in the number of words that need to be spoken for each software version compared to a standard text editor

From this graph it can be seen that all features provide some benefit compared to using a standard text editor and that the benefit when using all features is considerable, averaging 34%. Keyword replacement provides the greatest benefit, followed by white space removal and then constructor replacement. The different methods of name creation and use also reduce the number of words that need to be spoken since, even with none of the other functions working, there is still an improvement compared to using a standard text editor.

No results for version D and constructor replacement are shown since it does not provide this feature.

Figure 16: Graph to show how close the number of words spoken is to the target value of 1 word per written word

The use of all of these different functions means that the number of words that need to be spoken goes from an average of 71% of the optimal number of words to 94%. The optimal number of words being the number of distinct words in the programme, where a distinct word is defined as word that is surrounded by white space e.g. ‘myMessage’ or ‘print’ or a punctuation character.

These graphs give an indication of the overall improvement that could be expected when dictating a program since the test programs were structured to give the same approximate ratio of number of names, constructs and replaceable keywords as would be found in a real−life programme. However every programme is different so it is important to look at the actual improvements that each feature provides rather than the overall improvements, this is done in the following section.

Keyword replacement

Each line of the test programmes that contained text that would be affected by keyword replacement was analysed to find the improvement that having keyword replacement on would provide.

Figure 17: Graph to show the % improvement for each version of the CodeTalker for keyword replacement

The results for each software version are similar enough (only varying by 2.7%) to conclude that keyword replacement provides the same level of improvement in each. Since the same method of replacement is used in each version this is as would be expected.

The improvement provided by using keyword replacement is very high, averaging 37%. This indicates that, if the actual improvement was of the same magnitude, it is a very useful tool in reducing the time taken to programme by voice.

White space removal

Figure 18: Graph to show reduction in the number of words that need to be spoken when dictating code that requires white space to be removed

The results for each software version are similar enough (only varying by 1.6%) to conclude that white space removal provides the same level of improvement in each. Since the same method of removal is used in each version this is as would be expected.

The improvement provided by using white space removal does not seem overly high, averaging 8.9%. However as mentioned in the introduction this analysis compensates for the fact that Dragon Dictate learns to remove white space from before semicolons and closing brackets. If voice recognition software were used that did not do this then the benefits would be much higher. In the case of white space removal before semicolons every line of code would be affected.

Constructor replacement

Each line of the test programmes that contained text that would be affected by constructor replacement was analysed to find the improvement that using constructor replacement would provide.

Figure 19: Graph to show the improvement for each version of the CodeTalker for constructor replacement

No results are shown for version D since it does not provide constructor replacement.

As for keyword replacement, the improvement found by using constructor replacement is very similar across the three software versions to the extent that it can be concluded that each version provides the same level of improvement. This is an interesting result since constructors contain variable names and each version provides a different method of dictating variable names. It is not obvious that the improvements should be the same across all versions.

The percentage improvement depends on the constructor being used. Not surprisingly the higher the amount of punctuation in the constructor, the greater the improvement found when using constructor replacement since this punctuation is inserted automatically for the user.

Even for the shortest constructor, the while loop, the percentage improvement in words required to dictate it (17.5%) is still significant enough to make it worthwhile including.

Most programmes contain a variety of the constructors listed, which would indicate that the improvements found should be an average of the improvements obtained for each constructor. The average across all software versions is 29%.

Name Creation

Creation in this sense means the number of words that need to be dictated to get the name of a variable to appear in the editor, the first time it is declared. In the case of versions A and B this requires creating a new name in the list of available names and then referring to the position of the name in the list to get it written in the editor.

1. 2.

3. 4.

Figure 20: The four stages in creating a name using Version A. Version B uses the same method but requires a '~' after 'variable 1' to indicate that it should be replaced.

When subsequently using a name, it is only the third and fourth steps that need to be repeated.

For version C it requires specifying whether you are about to dictate a variable, function or class name and then dictating the words that make up the name. Version D on the other hand automatically detects when you are trying to dictate a name by looking at the context of the text. In the cases where it cannot tell the user can define what they are doing by saying 'variable', 'function' or 'classname'.

1. 2.

Figure 21: The two stages in creating a name using Version C

With version C the same two steps need to be repeated every time the variable name is used.

Figure 22: Graph to show improvement in words required for name creation and use with each software version compared to a standard text editor

The average values shown are the average of the improvements found during creation and use. This is the improvement that would be seen if a name were defined and subsequently used once. Generally a function, class or variable names will be used more than once in the course of writing a programme. The weighted average tries to give a better indication of the actual improvement by giving the improvements seen if a name is created and then used four times.

Though versions A and C initially appear to give similar results this is not truly the case since Version A requires more to be said when initially creating the name than C but requires far fewer words for each successive reference. Version A would therefore be quicker where a small number of names were frequently used and Version C more suited to a programme where a large number of names were infrequently used. This is clearly shown by comparing the weighted average and average results. Similarly version D provides far higher improvements when creating

names but the improvements become less marked compared to version A the more times the name is used. This is because, again, this method requires the entire name to be dictated each time it is used.

Conclusions

From the results it can be seen that all the features provide some benefit. Keyword replacement offers an average improvement of 39% over dictating the text that the keyword is replaced by and constructor replacement offers an average improvement of 29%. This analysis clearly indicates that both these functions should be included in an editor designed to improve programming by voice. While the results for white space removal are less impressive, giving an average improvement of 10%, this is still a significant improvement and, as discussed earlier, may provide even greater improvement with other voice recognition systems that are less adaptable. This analysis would indicate that this functionality should be included.

Both methods of name creation and use provide some improvement, although version B requires that a name be used at least twice, and version A at least once, for the time required to create it to be outweighed by the improvements seen by subsequently using it. The improvements found here are less than those found with keyword and constructor replacement. An exact figure for improvement cannot be given since it depends on the number of times a variable is used as to the improvements seen.

The effect of using keyword replacements far outweighs the benefit of any of the other features since keywords tend to appear more often than constructors and the percentage improvement is far higher than for name creation and use. This can therefore be seen as a key feature for reducing the work involved in dictating code.

In terms of the software version, D clearly provides the greatest benefit.

5.2 User Evaluation

Method

Three versions of the software (A, B & C) were evaluated using the within groups method where each subject evaluated each version.

The purpose of the evaluation was to find out:

- How the different methods of name creation compared in terms of efficiency and ease of use
- How efficiency was improved by the use of
    - keyword replacement,

    - constructor replacement,
    - white space removal, and
    - name creation
- How useful the user perceived the above functionality to be
- How useful the user perceived syntax colouring and line numbering to be.
- How the actual improvements found compared to the expected improvements predicted by the keystroke analysis

Ten subjects were given the task of dictating the set of three programmes given in Appendix A. Each subject dictated each program using a different version of the software and with certain features of each version switched on or off. The time taken for the user to dictate each line of code was noted in order to determine how much each feature improved efficiency. Each programme was also dictated using a standard text editor.

The use of different versions allowed the different methods of name creation to be evaluated. Switching functionality on and off allowed keyword replacement, constructor replacement and white space removal to be evaluated. In each case only one function was switched off so its effect could be directly measured.

Comparing results from dictation of one programme directly to another has difficulties since you are changing both the software version and the functionality being used. Dictating each programme using a standard text editor as well gives a benchmark to compare each programme to. In addition the time taken by two people to dictate the same line of code cannot be directly compared since each person dictated at different speeds. Instead the percentage improvement found by using the CodeTalker over a standard text editor for each person was compared.

The user also had to complete one class with the line numbers off and one with no syntax colouring in order to be able to evaluate how useful they found them. It was assumed that these features, while useful, do not greatly improve the efficiency of the programming task and hence they were not considered in the main analysis.

The evaluation was carried out in three stages, each of which lasted approximately an hour and a half. The evaluation was broken into stages partly to prevent boredom and also since it was found that Dragon Dictate recognises speech well for the first hour or so and then starts to deteriorate. The reasons for this are not known. Breaking the evaluation into sessions helped to alleviate bias caused by this and reduced frustration with the voice recognition software.

Stage 1: Dragon Dictate training

The user was taught how to use Dragon Dictate. The user then trained Dragon Dictate to recognise all the words required to dictate the three classes and to control the CodeTalker. Specific training of the words required meant that the same level of recognition was achieved for dictating the code as would take several sessions using the

software for general use. This training was intended to familiarise the user with Dragon Dictate and remove bias associated with inaccuracy in voice recognition.

Stage 2: CodeTalker training and first evaluation

As the evaluation was to find out how effective each method was any bias due to learning how to use the software needed to be removed. To do this the user was taught how to use each of the versions of the CodeTalker and given time to write a few sample classes.

The user was then asked to dictate the first class and timings for this recorded.

Stage 3: Final evaluation and user comments

The user was asked to dictate the second two classes and timings were recorded. The user then filled in a short questionnaire evaluating how effective they found each function to be and what their overall feelings and views on the CodeTalker were.

Throughout the evaluation the users were observed to see how they interacted with the voice recognition software and the CodeTalker and how this interaction affected their ability to program by voice.

To remove bias due to the order in which classes were dictated each user was given a different order in which to dictate the classes, and the order in which different functions were switched on and off was also rotated. Thus no one subject performed exactly the same order of evaluation as another.

Assumptions made:

- No bias due to learning to use the CodeTalker since the users were already trained
- Dragon Dictate recognised the user's voice at the same level of accuracy throughout the evaluation

Results

1. Users perception of the software

Usefulness

Users were asked to rate how useful they found each function in improving the programming task on a scale from 1 to 9 where 1 = nearly impossible to use with the feature, 5 = having the feature made no difference and 9 = nearly impossible to use without the feature.

Figure 23: Graph to show how useful the users perceived the different functions to be

All features were perceived by the user to improve the programming task. Functions found to be particularly useful were the keyword replacement, line numbers and the ability to dictate commands into the editor rather than access menu options (on−line parsing).

Ease of use

Users were asked to rate how easy each function was to use on scale from 1 to 9 where 1 = very difficult (had to ask for help several times) to 9 = intuitive.

Figure 24: Graph to show how easy the users perceived the different functions to be to use

Ease of use varied far more than the usefulness. Functions regarded as less easy to use were either those that required more cognitive thought (creation of names using lists, dictating constructors such as a for loop) or ones where they had less control over the actions of the editor. This included when the CodeTalker parsed each word of a command as soon as it was spoken rather than waiting for a terminating '~' to be said by the user and parsing constructors word by word.

Preferred methods

Users were then asked to specify the method that they preferred to use to create names, use commands and use constructors.

Figure 25: Pie charts to show user preferences

100% of users preferred the method of constructor parsing where the CodeTalker waits until a terminating character is entered before parsing as opposed to parsing word by word. For this reason a pie chart for constructor parsing is not shown.

This clearly shows that the users preferred to have more control over the CodeTalker by forcing it to wait to parse a command or constructor until a terminating character is seen. Although dictating the complete name of a variable, class or function may seem the more natural method the majority of users preferred to dictate names by referring to their position from a list of defined names.

2. Results from using the software

All results carry a maximum error of 2% and an average error of 1.49%.

The reason for this is that, on average, the user stopped just over four times during the dictation of the programme for reasons not associated with the task. These reasons included coughing, sneezing, stopping for a drink of water or because they were having problems getting Dragon Dictate to recognise a certain word. Since the aim of the project is not to investigate how good voice recognition software is, the time spent correcting Dragon Dictate is not included in the timings. Including the time taken to correct errors would skew the actual time taken to write a line of code and therefore make it much harder to interpret the effect that the functionality of the CoderTalker has on improving the programming task.

The shortest time taken to dictate a programme was 202 seconds and the average 312. The range of times is due to the different program set−ups being used. All times were taken to an accuracy of one second. To reduce inaccuracies, whenever the users re−started dictation they did so at the start of one second. Thus the only inaccuracies were the times noted when the user stopped which were accurate to ± 0.9 second. As the average number of stops (including completing the programme) was 5.16 this gives a total error of ± 4.65 seconds.

This is 1.49% of the average time taken to dictate a programme and 2% of the minimum.

Overall Improvements

Figure 26: Graph to show overall improvements for each version of software

The graph above shows the actual percentage improvements found when dictating code using each of the three software versions when compared to the same user dictating the same code using a standard text editor.

Each version has constructor and keyword replacement on and white space removal on.

It can be seen that Version C gives by the far the best improvement in time with Version A giving the lowest, though all versions do provide an improvement. From the keystroke analysis it would be expected that Versions A and C would give similar results. However the main problem with Version A is that the accuracy of the voice recognition software is not 100%. As version A parses everything immediately, rather than waiting for a terminating character as Version C does, this means that any errors in dictation immediately result in a warning being displayed to the user.

This is not generally a problem but becomes so when dictating constructs which have a very specific format. All the users that had to dictate a ‘for’ loop found it impossible and only one person managed to successfully dictate a while loop. After trying for two minutes users were able to give up and pass on to the next line of code, however this two minutes is still included in the total time taken and so reduces the improvements found by using Version A.

While this cannot be ignored it is not a fault of the CodeTalker but of the voice recognition software. For this reason the improvement found when ignoring the lines containing constructs is shown as Version A* and is around 27%.

This indicates the level of improvement that could be expected when using Version A when the accuracy of voice recognition systems has increased from current day values (~96%) to close to the 100% mark.

The above graph gives the overall improvements provided by a CodeTalker with all functionality active compared to using a standard text editor, all of which improve the time taken. The graphs that follow look at the contribution from each of the different functions (keyword replacement, constructor replacement and white space removal) to achieving these improvements, i.e. which functions are the most beneficial and are they beneficial?

In the same way that the keystroke analysis looked at the actual improvements for each line of code affected by keyword replacement and constructor replacement the same was done for the actual results.

Keyword replacement

Figure 27: Graph to show actual improvements found by using keyword replacement

The first graph shows the overall improvement in using the CodeTalker with no keyword replacement to using it with Keyword replacement to dictate an entire programme.

The second graph looks at the improvement in time for dictating the specific lines of code that contain keywords.

Both graphs indicate a high level of improvement when using keyword replacement. The fact that the improvements in overall dictation are greater for versions A and B indicate improved speed of name creations using these two versions over Version B (since the entire programme is being considered).

By considering only the lines that contain keywords the improvements can be more clearly seen. The improvements are similar across the three programmes with the average improvement being 27%. This is a significant improvement in time and indicates that keyword replacement is a key tool in reducing the time that it takes to write a programme by voice. This usefulness is reflected by the views of the users, where the average rating for the usefulness of keyword replacement was 8.5 (9 represents the view that the programming would be nearly impossible without the feature).

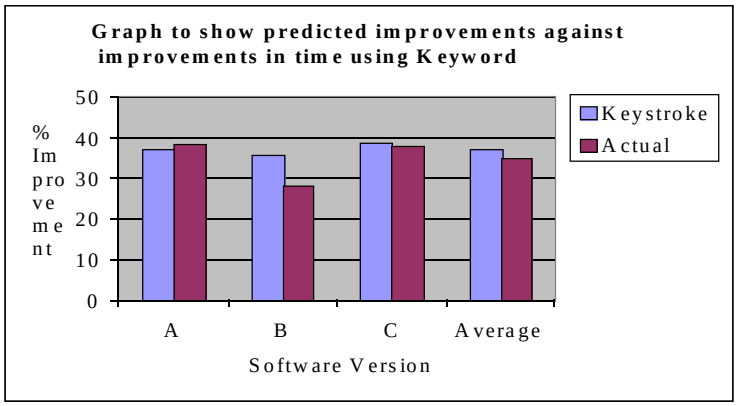

Figure 28: Graph showing the predicted improvement in time compared to the actual improvement found

This graph allows us to see how easy the feature is to use. The actual improvement found should only differ to those predicted by the keystroke analysis if there is some delay due to cognitive activity. The most likely reason for the user having to stop and think would be if they were unsure how to use the feature and had to think about what action to take next. Therefore if the actual results are close to those predicted we can conclude that the feature is easy to use since the user does not spend long thinking about it.

The graph above shows that the actual improvement in speed is very similar to that predicted by the keystroke analysis. For versions A and C the difference is within 1%. Version B is not quite so accurate, the difference between predicted and actual results being 7.5%. Since all three versions use the same method for keyword replacement the difference between the actual and predicted results should be the same for all three. The results for version B can only be described as an anomaly. The average variation is still only 2.5% and given that all results carry an error of 1.5% this is close enough to the predicted results to conclude that the method used for keyword replacement is highly effective and easy to use. This tallies with the user's average rating of 8.5 out of 9 for ease of use.

Constructor replacement

Figure 29: Graphs to show the actual improvements found by using constructor replacement

The first graph shows the overall improvement in using the CodeTalker with no constructor replacement to using it with Constructor replacement to dictate an entire programme. The second graph looks at the improvement in time for dictating the specific lines of code that contain the constructors. The negative results seen in both graphs indicate the fact that using constructor replacement actually slowed the programming task down when compared to using a standard text editor.

Some explanation is needed to explain the rather contrary results shown by the two graphs. The second graph shows that using constructor replacement speeds up dictation by 36% and 18% when using Versions B and C respectively, however the improvement in the time taken to dictate the complete programme is slower by 1% in both cases. While the improvements may sound large the fact is that the actual time to dictate a constructor is small, averaging 14.6 seconds with replacement to 11.6 seconds without it for versions B and C. This is only a difference of three seconds, around 1% of the time that it takes to dictate the entire programme. So while constructor replacement provides an improvement on the time to dictate a constructor it has little impact on the total time. The fact that the results overall show a 1% decrease rather than a 1% increase can be attributed to the ±2% error found in the results.

The results for programme A are interesting. Since the method used is slightly more efficient than that provided by Version B you would expect to see at least a small improvement (as predicted by the keystroke analysis) rather than a very large decrease. The reason for the very different results is the fact that version A parses each word immediately as it is spoken and will display an error message as soon as a mistake is found. Unfortunately voice recognition software is not yet perfect and cannot provide the accuracy needed to dictate a very specific set of words. As soon as one word is incorrect, the user is stuck trying to correct the voice recognition software while dealing with errors being displayed by the CodeTalker as well. This quickly becomes frustrating and means that dictating a complex constructs such as a ‘for’ loop was found to be an impossible task.
While the theoretical benefits of Version A may be great, until the accuracy of voice recognition software improves parsing each word as it appears is not a good method to use.

Figure 30: Graph showing the predicted improvement in time using Keystroke analysis to the actual improvement found for constructor replacement

Looking at the comparison between predicated and actual improvements the difference between the two for version A clearly illustrates the difficulty involved in using it. The other two versions are similar though, with version B being 7% slower and version C being 10% faster. The difference here can be explained by the fact that with version B the position of the variable name in the list needs to be found which takes time, whereas with version C the complete variable name is dictated so the cognitive time to find the name is not present. Hence version C is closer to the predicted time.

Apart from the method used to define variable names both versions B and C use the same method of constructor replacement – getting the user to dictate the complete text and only parsing the text when a terminating character has been spoken. On termination the relevant punctuation is inserted, numbers are converted from text to actual numbers (e.g. one −> 1) and variable names are created. The fact that version C is faster than predicted and version B is only slightly slower (which would be expected due to time taken to locate the variable name) indicates that this method is an optimal one and that this method of constructor replacement is easy to use.

On the other hand the method employed by version A, parsing each word as it is spoken, is clearly not easy to use. This is reflected by the views of the users who rated the method used by versions B and C as 7.9 out of 9 for ease of use and the method used by version A as only 6.2 out of 9. Given the choice all the users said that they would prefer to use the method given in versions B and C.

White Space Removal

Figure 31: Graph to show the % improvement in time when using white space removal for each version of the Code Talker

At first glance there may appear to be a difference between the three versions, however the variance across them is only 0.4%. This means that the improvements in each version are similar enough to be considered identical.

This is as would be expected, since there is no difference between each version as to the amount if white space that needs to be suppressed. Though the benefits would be even greater in situations where the voice recognition software does not remove white space before semicolons or closing brackets there is still a clear benefit even when the use is somewhat limited (the reasons for this are discussed in the Keystroke analysis, section 5.1).

The average improvement is exactly the same as that predicted by the keystroke analysis, indicating that the white space removal method is both optimal and easy to use. This corresponds to the users feedback – all users gave white space removal 9 out of 9 for ease of use (indicating that it was intuitive and needed no explanation). They gave it an average of 7.5 out of 9 for usefulness, indicating that they thought it helped the programming task significantly.

Name creation and use

Figure 32: Graph to show the improvements in time for name creation and use for the different versions of software

The negative results seen for creation indicate that creating a name using this method is slower than simply dictating the name in a standard editor. This reduction in efficiency is outweighed by the improvement seen in future dictations of the name shown in the ‘use’ section.

As discussed in the Keystroke analysis section Version C gives the best improvements for creation and Version A the best for use. This is due to the different methods used as C requires dictating the entire name each time the variable is used whereas A only requires full dictation once, the variable is then referred to as ‘variable x’ where x is it’s position in the list of available variable names.

The average of the creation and use improvements gives the improvement seen if each variable was declared and used once. This is rarely the case in real life since variables are generally used several times. The weighted average attempts to give a more realistic idea of the advantages of each method and gives the overall improvement

when creating a variable and using it four times. The weighted average clearly shows a benefit of version A over C.

From the graphs below it can be seen that the actual improvements in time are greater than those predicted by the keystroke analysis in all cases except for name creation using version B. This is surprising since you would expect the actual times to be slower not faster. While they are all effective version A provides the biggest improvement for creation and version C the biggest improvement for use.

Figure 33: Graphs to show the actual improvement in time compared to that predicted by the keystroke analysis for name creation and use

It would be expected that versions A and B to show less of an improvement against predicted time than version C since they both require the user to locate the name that they want from a list of names, as locating the name takes time this would reduce the improvement predicted by the keystroke analysis. Version C does not have this task and should therefore reflect the predicted time more closely.

Averaging the difference between expected and actual improvements for name creation and version A comes out as the best (averaging 8.5%), version B is exactly as predicted (0% difference) and version C is in the middle with 6.5%.

This would indicate that versions A and C are the easiest to use with version A being slightly superior. The users actually found version C slightly easier to use, giving it an average ease of use rating of 8.2 out of 9 compared to 7.4 out of 9 but the majority (70%) said that they would prefer to use version A over version C if given the choice. Once again their views correlate closely to what actually speeds up the task the most and appears to be the easier method to use

5.3 Expert Evaluation

The aim of the expert evaluation was to determine how effective the method deployed by version D of the CodeTalker was. This version was developed after the HCI evaluation as a result of new ideas. As the HCI evaluation conclusively showed the methods used for white space removal and keyword replacement to be effective they were left unchanged in this version, however the methods for dictating variable, function and class names are very different. In addition it does not try to provide constructor replacement as the results of the HCI evaluation indicated that this was a hindrance rather than a help to the programming task.

It works on the simple principle that any text that it not a language keyword, a string or number is a name. By looking at the surrounding text and having a blueprint of the structure of class, function and variable definitions the CodeTalker is able to automatically determine whether you are dictating a name and, if so, when you have finished. Once you have finished dictating the word the CodeTalker automatically replaces the text with the created name. By dictating a terminating character you can force the Code Talker to concatenate the words at any point, this deals with the case where you cannot determine between the end of one word and the start of the next.

e.g. private TabbedPane myPane;

Theoretically this should be a faster method since the number of words that you need to say is reduced. It is only occasionally that you need to specifically terminate the name by saying ‘circumflex’ and only rarely that you need to say ‘variable’, ’function’, or ‘classname’ to indicate that you are about to dictate a name. This theory is backed up by the keystroke analysis, which gives this method the highest percentage improvement over using a standard text editor.

As this version was developed well into the project there was not enough time to perform a full HCI evaluation of it. However an expert evaluation should be able to give a good indication of how effective the method is.

Method

The same classes were dictated as for the HCI evaluation. This meant that the results between the two could be compared.

The effectiveness of white space removal and keyword replacement have already been evaluated in the HCI evaluation and are not reconsidered here. Only the overall improvements and those found for name creation and use are evaluated.

Results

Figure 34: Graph to show the overall % improvement in time taken to dictate the set of programs when compared to using a standard text editor

This graph shows the overall improvements found when dictating complete programmes using version D compared to the improvements found using the other versions. The top of each line indicates the maximum improvement, the bottom of the line the minimum and the central black bar the average. This graph shows that as well as the average improvement found being greater than all other versions it was also greater than the maximum improvement found with all other versions. This clearly indicates the superiority of this method over the others.

It could be argued that an expert evaluation should produce better results and this therefore skews the results. However every person who was involved in the HCI evaluation had similar training and was of the same competence as the expert evaluator for this version, which means that this should not be the case. The results from the two evaluations should be directly comparable.

The direct improvements found when dictating lines of code containing variable, class and function names is now considered.

Figure 35: Graph showing the % improvements when using Version D to create and use names against all versions

The method used by this version is clearly an effective one since it provides the greatest improvement across the board. Since both creation and use of names requires you to dictate the full text of the name you might expect that the results for the two should be similar. However this does not appear to be the case. On closer investigation this is not an unusual result, since when a variable, function or class is created its creation is preceded by dictation of the words private, public, protected, void or a class name of some description. It is easy for the CodeTalker to recognise that you are about to dictate a name and so the user never needs to expressly tell the CodeTalker what they are trying to do. On the other hand, when using a name it is not always obvious to the CodeTalker, this is especially true for variables. For example, consider the line:

myMessage = “hello”;

When you start the a new line the CodeTalker makes no assumptions about what you are about to dictate so the user needs to say ‘variable’ to indicate that they are about to dictate a variable name. This increases the number of words that need to be spoken, thus slightly lowering the efficiency.

Figure 36: Graph to show the improvements predicted by keystroke analysis compared to the actual improvements found

The actual improvements found exceed those predicted by the keystroke analysis by an average of 12.9%. This indicates that this method is a highly efficient and intuitive one to use. This is not overly surprising since for the majority of the time the user does not have to think about how to go about creating the name, the CodeTalker simply does it for them. This method was felt by the expert evaluator to be the simplest to use.

5.4 Speech Directed Interface

Although this project concentrates on improving the task of programming by voice the CodeTalker was also designed to make controlling the different menu options easier. To perform a task, such as closing a file, with Dragon Dictate, requires entering the command mode by saying 'Command Mode' and then dictating the name of each successive menu option. E.g. to close a file you would need to say 'command mode, file, close'. While using drop down menus is easy with a mouse, having to name each menu option and enter another mode is tedious. The more complex menu operations which contain dialogue boxes range from being tedious to use (requiring saying 'Tab key' to navigate around to the button that needs to be clicked) to impossible, for example selecting a tabbed pane.

Instead of requiring users to access drop down menus, the CodeTalker looks at each word dictated to see if it is a command. If it is, the text of the command it is highlighted as it is dictated to give a visual cue to the user. On completion of dictation the text is removed from the window and the command performed.

This evaluation looks at whether this method improves upon the other method in terms of speed and ease of use.

Method

Each menu option was accessed first using the on−line method whereby the command was dictated straight into the editor and then by accessing the drop down menu's. The time taken for each method was noted.

Dragon Dictate accesses menus by 'typing' the shortcut keys for that menu option. For example, in Microsoft Word saying 'Command Mode, File, Save' would send the keystrokes 'Alt+S' to Word, causing it to save the file. This is fine for programmes that have the same standard shortcut keys but, if not, means that Dragon Dictate needs to be taught which keys to 'press'. In addition to this Dragon Dictate does not seem able to cope with interfaces other than standard windows ones. Despite the fact that the CodeTalker has the same shortcut keys as any Microsoft software and pressing 'Alt+f' directly opens up the File menu, setting up a Dragon macro to perform the same keystrokes does not bring up the menu. Despite repeated attempts at trying to solve this problem I could not see why this was happening. Sending the F10 key to the editor gives access to the menu options but this is an incredibly tedious method, requiring tabbing along to the relevant menu.

Since this is obviously a shortcoming of the CodeTalker software the times taken to access menu options via on−line parsing were compared to the times taken to access the same menu options using a standard text editor that responds to Dragon Dictate's commands. Only the menu options common to both software packages were considered.

Results

Figure 37: Graphs to show actual and percentage speed−up when using on−line command parsing compared to accessing the drop down menus

From the graphs it can be clearly seen that on−line command parsing is a faster method to use. The average reduction in time for invoking a command was just over six seconds, an improvement of 54%. The on−line method

was quicker for every command that was invoked, not just on average. This can be seen by the fact that both the maximum and minimum improvements are positive, i.e. there was always an improvement even if was only by one second.

What these results do not show is that it was impossible to turn syntax highlighting on or off using the standard text editor. To do this required bringing up the configuration menu dialogue box, shown below, selecting the pane marked ‘syntax highlighting’ and then changing the value of a check box.

Just to navigate to the Syntax highlighting pane required saying ‘Tab Key’ twenty−two times followed by saying ‘left arrow’. Once on the correct pane it took a further six tabs to reach the relevant check−box. However, after all this, it was found to be impossible to change the value of the check box.

Figure 38: Screen shot of the configuration menu. Changing to the ‘Syntax Highlighting’ pane requires dictating ‘Tab Key’ twenty−two times.

5.5 Conclusions & Recommendations

The HCI investigation and expert evaluation aimed to find out whether white space removal, keyword replacement and constructor replacement actually improved the task of programming by voice and how good the methods used where. It also looked at whether providing methods of creating variable, function and class names improved the task and which method was best.

A summary of the improvements found by users is show in the graph below.

Figure 39: Summary of the results from the HCI and Expert Evaluation

Keyword replacement is obviously the most significant in terms of improving the speed of the programming task, the actual improvements found were almost exactly as predicted by the keystroke analysis indicating that the method employed was simple and easy for the users to use. The users own views backed this up.

While constructor replacement may seem like a good idea the results of this evaluation would indicate that it is not. The actual improvements are small (in the region of seconds) but the time taken for the user to remember the format of what they need to say removes any overall gains. Since braces and brackets can be easily put in using keyword replacement (replacing the word ‘braces’ with two braces and a new line between them, leaving the cursor inside the braces) it is not as time consuming to write a do and while loop as may be expected. ‘For’ loops are somewhat tedious but are not greatly improved by using constructor replacements.

Evaluating constructor replacements also brought to light the problem of parsing each word immediately. If you are going to parse anything that requires a very strict syntax then you should allow the user to completely dictate the sentence and not parse it until they have said some terminating character. This allows them to correct any mistakes that the voice recognition software may have made. The reason for this is that voice recognition software

is simply not accurate enough at present to allow parsing of each word as it as spoken. The resulting problems when errors are detected are both irritating to the user and harder to cope with programmatically. This is well demonstrated by the huge reduction in performance (29%) when trying to parse constructors word by word and the fact that not a single user would prefer to use this method over waiting for a terminating character.

Little has been said about the methods used to perform commands such as saving files, changing windows etc. Here the user was also given the choice between parsing each word as it was spoken or waiting for a terminating character. Although the sentences being spoken were short (nearly all commands are two words long, a few are three words) 80% of users still preferred to terminate the command themselves, and thus retain greater control over the CodeTalker. While parsing each word as it is spoken is theoretically the best method to employ until the accuracy of voice recognition systems improve it should not be used.

Providing on−line parsing rather than making the users navigate normal drop down menus by voice provides a high level of improvement, averaging 54%. This is obviously a good alternative and should be provided in any programme designed to be controlled by voice, not just one designed to allow programming by voice.

White space removal, while not as high as keyword replacement, still provided a good level of improvement, averaging just over 9%. This improvement was exactly as the keystroke analysis predicted, again indicating that it was an effective method and was found by the users to be intuitive.

Overall the best method of name creation and use was that used by version D, where the CodeTalker used the context of the words to automatically detect when names were being dictated. The fact that it improved dictation speed by over 15% more than any other method clearly shows it to be the most efficient to use. Comparison with the keystroke analysis also indicated that it was the simplest method to learn.

Recommendations

The following section describes the features that an editor designed to support programming by voice should and should not have. The different features are described in order of importance.

Keyword replacement

The user should be able to define specific words that are replaced by other text, e.g. replacing ‘print’ with ‘System.out.println();’. As well as defining the text to replace the word with they should also be able to define the final position of the cursor within the text.

To perform the replacement the editor should wait until it sees white space, indicating the end of a word, then read in the word that precedes the white space and see if it should be replaced. If it should the text should be removed from the editor window and replaced with the relevant text and the cursor moved to the correct position.

A word should not be replaced until white space is seen since that leads to problems when a word contains a keyword, e.g. ‘printing’.

Appendices B and C detail the basic set of replacements that should be provided for HTML and Java.

Variable, Class and Function names

The editor should be able to use the context in which the text is being dictated to decide when a user is trying to dictate a name. When it is not possible to use context to decide, the user should be able to define variable, class and function names by specifying the name type followed by the words that should form the name. On seeing a specific name terminating character (for instance a semi−colon) the editor should remove the words that form the name from the editor window, concatenate them and display this completed name.

White space removal

The editor should check that there is no stray white space that may make the code invalid to a compiler. Any stray white space should be removed immediately without prompting the user. Whenever the editor reads in a punctuation character that should not have white space either before or after it, it should check the previous or next character and, if it is white space, remove it.

e.g. System. out. println () ; should have white space removing to read: System.out.println();

Commands

Rather than making the user use the standard drop down menus the editor should allow the user to dictate commands directly. Once the command has been dictated the text of it should be removed and the action requested performed. While the command is being dictated it should be highlighted in some distinct colour so that the user has an immediate visual cue. The full text of the command expected should be displayed in the menu bar at the bottom of the editor to act as a prompt.

Unless the voice recognition software being used has near−perfect accuracy a command should not be parsed until the user has completely dictated it and signified this by dictating a terminating character.

If, for some reason, on−line parsing cannot be supported and the user has to use a drop down menu it should be designed in a way that makes it accessible by voice. In general this means that there should be no more than one sub−menu within each menu and dialogue boxes should not be used.

Line numbers and syntax colouring

While syntax colouring does not improve the programming task per se it allows for easier reading of code and is a general standard that an editor designed to allow programming by voice should provide.

Line numbers are almost essential for allowing quick navigation. The editor should provide a command to allow the user to jump to a specific line and display line numbers alongside the text of the programme.

Constructor replacement

Recognising specific programming constructs such as for, while and do while loops and automatically inserting the required punctuation does not significantly improve the programming task. Like command parsing a construct should never be parsed word by word unless the voice recognition software provides perfect recognition, instead the editor should wait for a terminating character.

Generally speaking constructor replacement it not a worthwhile method and there are other areas that time could be better spent on improving, for example providing debugging facilities by voice or improving navigation.

Relation to existing work

The results of this investigation show that existing work does help to improve the task of programming by voice but is not optimal. The method of name creation employed by version D, automatic recognition of names by examining the context, is the most optimal method but is not used by any of the software currently available. In addition no work has currently been undertaken in designing an editor specifically towards being controlled by voice, something that has been shown by this investigation to significantly improve efficiency not only of the programming task but of any task that requires using menus.

6. Overall Conclusions

While the CodeTalker does not solve all the problems associated with programming by voice it does significantly improve the task. With version D being used the improvements are enough to bring the number of words that need to be dictated close to one word per distinct word on the screen. In addition the new methods proposed within this project have been shown to improve upon software currently available.

The results of the HCI analysis and Expert Evaluation showed that to best improve the task of programming by voice an editor should provide, in order of importance:

- Keyword replacement: replacing specific words with certain phrases or punctuation marks to reduce the number of words that the user needs to dictate
- Automatic recognition of when a user is trying to dictate a variable, class or function name by analysing the context in which they are dictating text and concatenating the words dictated in a way that forms a valid and conventional name*
- Removal of stray white space that would result in the text being rejected by a compiler*
- Access to all functionality contained in any drop down menus by dictating text into the editor window*

Functions noted with a * are those that were developed during this project and are not currently available in the research domain.

This optimal functionality was implemented in version D of the CodeTalker and provides improvements in the region of 44% in the time taken to dictate a programme compared to the time it would take to dictate the same code in a standard text editor such as UltraEdit or Notepad.

Although other currently available programming by voice products already use keyword replacement the other three functions were developed during this project. Their increased efficiency over other methods currently used mean that version D provides a 15% extra improvement in time over all other software versions which use existing methods.

The software developed is robust enough to cope with dictation errors and gives the user a good deal of control over the programming environment, allowing them to adjust it to how they best like it. The architecture used to implement the features, while not necessarily optimal, has resulted in an editor that is fast to use, even with the computational demands of the voice recognition software. To put this into context version C was adapted to include one hundred commands to check for, rather than the standard eighteen, and five hundred keyword replacements. Syntax colouring and all other functionality was switched on. Even with these demands there was no perceptible slow−down, although Dragon Dictate was also running along side it. This scalability testing was

performed on a laptop computer with a 233Mhz processor and 64Mb of RAM, a configuration that would generally be considered sub−standard.

The modular architecture of the software means that it is straightforward for a Java programmer to add new functionality or to adapt the CodeTalker to some task other than programming. Using a single class to provide an abstraction between the details of the command being entered (coming from the editor window) and the classes called to invoke the action requested provides an elegant way of handling voice controlled software. By changing the link from the command processing class to the classes that perform the actions the functionality of the CodeTalker can be quickly, and completely, changed. Altering the link between the editor and the command processing class allows a completely different front end to be attached to the CodeTalker. The method used by the Command processing class of specifying a command in terms of a vector and parsing for specific types allows new commands to be quickly created and doesn't require the programmer to understand how the command parsing works. A programmer only needs to understand how to define the structure of the command and where to add the 'hook' to the class that performs the function.

Using the toolkit the author was able to develop a simple voice−controlled drawing package in just a two hours.

The HCI investigation also brought some interesting points to light. The close correlation between the keystroke analysis and the actual improvements found would indicate that this is a good method to use to estimate the improvements that can be expected. This is generally the case with keystroke analysis but did not mean that it would necessarily translate to a 'words dictated analysis' which is what was being done here. In addition the users themselves were very good at distinguishing the features that provided the most benefit and those that were not useful. The order in which they put the different features in terms of usefulness was exactly the same as the actual order found on analysis. Similarly the ease of use of each function reflected the correlation between the results predicted by the keystroke analysis and the actual results found. Generally any formal investigation should involve timing a large number of users performing a task to gain a real idea of the improvements that can be expected. However if time were short this information would indicate that a 'words spoken' or keystroke analysis along with an expert's view of how useful and easy to use a function is would provide a very realistic idea of the improvements that could be expected

Another very interesting point is that of accuracy. While this project has focused on the aspect of bringing programming to those that are unable to use a keyboard or mouse programming by voice has an interesting implication for all computer users. The reason for this is that, while voice recognition systems occasionally recognise words incorrectly, they are generally very accurate (around 98%). More importantly, when they do recognise a word correctly they never misspell it, unlike humans. Writing computer code requires absolute accuracy in the spelling and syntax of names and a common reason for a program not compiling is that a word is misspelled or punctuation, such as a closing bracket, is missing. Since programming by voice automates much of the work involved in adding in punctuation (e.g. replacing keywords such as brackets with a pair of brackets)

errors due to missing punctuation are reduced. Short tests using the author as an expert evaluator have shown that voice recognition systems make far fewer mistakes than humans when writing code, even when the person is an experienced typist. This means that code written by voice is likely to contain fewer mistakes and therefore require fewer corrections before it correctly compiles.

Tests showed that the speed of voice recognition software only needs to improve slightly to make it as fast to programme by voice as by keyboard. Coupling this with the fact that the code produced will be less error prone and so compile faster means that this may well be a more efficient method. Finally, the most frustrating aspect of programming by voice is trying to navigate and switch windows quickly. If you are an able−bodied person you can simply perform these tasks using a mouse, thus greatly reducing frustration. Companies are always looking out for new ways to improve efficiency and this may well be one of them.

7. Future Work

Although the editor described improves the programming task there are still areas that can be improved upon. These include:

- Developing a method to allow users to adapt their personal style to voice. Saying 'Tab' continuously is very tedious and slows programming down. An improvement would be to enter a 'Tab 2' or 'Tab 1' mode and indent by the specified number of tabs on each new line until the user specifies otherwise. Personal styles can greatly slow a user down but the HCI evaluation showed that it is very hard to change. As well spaced code it far more readable this is an important feature for both speed and readability.
- Automatically ending lines. Many languages require that each line containing code be completed with a specific character, often a semicolon. Efficiency could easily be improved by automatically adding in this punctuation each time a new line is dictated on a line containing code.
- Improving navigation. Although line numbers help hugely users still had problems moving to a specific point in a line. Current dictation software does not provide very elegant methods to overcome this. One method proposed would be to display the text in a grid, i.e. with column as well as row numbers. The users could then refer to a grid reference and move to the exact position that they are after in one command.

A more important project would be to extend the ideas developed in this project to provide a complete programming environment i.e. an editor, debugger and compiler, such as that provided by Microsoft Studio. Interestingly a bug in Microsoft studio means that, although you can dictate text and invoke the compiler by voice, you cannot run the debugger, since it causes the programme to crash. Development of such an environment has been suggested by the Archimedes Project [**10**], a project designed to give disabled people full access to all areas of computing, but has not developed beyond saying that it would be a useful tool.

This project has tried to make programming in current commercial languages possible, however a completely different approach that could be tried is to develop a language specifically for programming by voice. Key features of such a language would be a tolerance to white space before or after punctuation and a reduction in the amount of punctuation that needs to be dictated. Such a language could well be radically different to ways in which we programme now, however the reduction in typing errors when using voice recognition software may well make such a method faster than using a keyboard.

Whatever solutions are developed it should always be kept in mind that many design decisions within this project were made to make the software robust to the voice recognition software incorrectly recognising words. As voice recognition software improves and their accuracy increases methods that were not recommended may become possible and other solutions may present themselves.

There are many other areas that can be explored where programming by voice is concerned, not just providing tools for the disabled. These include:

- Allowing dynamic creation of filtering rules for e−mail and internet content to be downloaded to your phone by calling a central computer and dictating the rule set
- Changes to, or interactions with, programmes in areas where computers are available but using a keyboard would not be possible e.g. invoking small test programmes to test repairs as they are carried out on the outside of a space station. The astronaut is already equipped with a headset but using a keyboard would not be practical.
- Performing mathematical calculations (such as those provided by Mathematica and MathCad). E.g. you may have a mathematical model to determine oil pressure at certain depths on an oilrig. Divers performing repairs can adjust the model according to what they find to work out whether the pressure at different points is correct or not.

Finally this project has focused on creating a voice controlled development environment, however there are many other tasks that are difficult to perform by voice. These include using drawing packages, presentation software such as PowerPoint and anything that relies heavily on a mouse. Projects in the future could use the toolkit of software developed during this project to write voice−controlled software that perform these tasks.

Bibliography

1. Larris, Richard. “Lucent Technologies adds speech−enabled directory searching to it’s enterprise Directory Solutions”. Sept 14 1999. http://www.lucent.com/press/0999/990914.bca.html

2. Kaye, S. “Computer and Internet Use among people with Disabilities”. March 2000. http://dsc.ucsf.edu/UCSF/spl.taf?_from=default

3. The USA Bureau of Labor Statistics, “Total Employment in 1998”. April 5 2000. http://stats.bls.gov/emphome.htm).

4. National Centre for Health Statistics, “Vital and Health Statistics”. May 10 2000. http://www.cdc.gov/nchs

5. Desilets, Alain. “Voice Code, Programming by Voice Toolkit”. http://ii2.ai.iit.nrc.ca/VoiceCode

6. Epstein, Jonathan. “Writing and Debuggin Code by Voice – A Proof of Concept’ May 6 1998. http://voicerecognition.org/developers/jepstein/JavabyVoice/index.html)

7. Rene, Thomas. “Demacs Macros”. ftp://ftp.cl.cam.ac.uk/a2x−voice/demacs.tar)

8. Lucas, Bruce. “Voice XML”. 1 Oct 1999. http://www.alphaworks.ibm.com/tech/voicexml

9. Janin,Adam. “Design of a Speech Orientated Programming Language”. http://www.icsi.berkeley.edu/~janin/sopl/cover.html

10. Bienstock, Hilary. “The Archimedes Project”. Aug 17 1999. http://archimedes.stanford.edu/index.html

11. IBM Computers, “ViaVoice Speech Application Programming Interface”. Aug 1997. http://compy.ww.tu−berlin.de/IBMVoice/SAPI/sapiref.htm

12. Dix, Alan, Janet Finlay, Gregory Abowd and Russell Beale. “Human−Computer Interaction”. 1993 Prentice Hall. ISBN 0−13−437211−5

13. Horrocks, Ian. “Constructing the User Interface with Statecharts”. 1999 Addison−Wesley, ISBN 0−201−34278−2

14. Macaulay, Linda. “Human−Computer Interaction for Software Designers”. 1995 Thomson Computer Press. ISBN 1−85032−177−9

15. Markowitz, Judith. “Using Speech Recognition”, 1996 Prentice Hall, ISBN 0−13−186321−5

16. Niederst, Jennifer. “Web Design in a Nutshell”. 1999 O’Reilly. ISBN 1−56592−515−7

17. Weinberg, Gerald. “The Psychology of Computer Programming” 1998 Dorset House Publishing. ISBN 0−932633−42−0

18. Bernsen, Niels et al, “Designing Interactive Speech Systems”, Springer 1998

19. Newman, William et al, “Interactive System Design”, Addison Wesley, 1995

Appendix A – Programme Set

All classes are shown with indentations to aid ease of reading, however indentation were not used when dictating.

Class Count Down

```
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class CountDown {

private int maxValue;
private int minValue;
private String helloString;
private int I;

CountDown(String name) {

maxValue = 10;
minValue = 0;
helloString = "hello";
this.sayHello(s);}

public void countDown() {
int i;
System.out.println("counting down");
for (i=maxValue;i>minValue;i—)
{
System.out.println(i);
}
}

private void sayHello(String name)
{
String returnString;

if (name.length() > 10) {
String upper = name.toUpperCase();
System.out.println("hello");
returnString = upper;}
else {
String lower = name.toLowerCase();
```

Class Count Up

```
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class CountUp {

        private int maxValue;
        private int minValue;
        private String name;
        private String myMessage;

CountUp(String s) {

maxValue = 10;
minValue = 0;
myMessage = "hi";
name = s;
this.sayHello(s);
}

public void countUp() {

int count = minValue;
Do {
char c = name.charAt(count);
System.out.println(count + c);
count ++;}
while (count < maxValue);}

private void sayHello(String s)
        {
        if (s.equals("hello")) {
        System.out.println(myMessage);}
        else {
        String lower =
name.toLowerCase();
        System.out.println("hello" + name
```

Class Factorial

```
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class Factorial {

        private int maxValue;
        private int minValue;
        private String name;
    private String errorMessage;

Factorial(String s) {
        maxValue = 10;
        minValue = 1;
        name = s;
        errorMessage = "error";
}

public void go() {

int count = maxValue;
        int currentValue = minValue;

while (count > minValue)
{ currentValue = currentValue *  count;
        count—; }
this.displayResult(currentValue);
}

private void displayResult(int i)
{
String value = Integer.toString(I);

if (i < 100) {
errorMessage = errorMessage.toUpperCase();
System.out.println(errorMessage); }
else if (i > 100) {
```

```
System.out.println(s);
returnString = lower;}
return returnString;
}}
```

```
+ lower);
}
}
```

```
value = value.toLowerCase();
System.out.println(I + value); }
}
}
```

Appendix B − Default keyword replacements for HTML

For HTML the replacement text is also the syntax that should be highlighted. Therefore all replacements should be syntax highlighted as well.

Text	**HTML replacement**	**Text**	**HTML replacement**
anchor	<a>	list item	<menu> * </menu>
body	<body> * </body>	menu	
head	<head> * </head>	anchor	<area>
html	<html> * </html>	area	<map> * </map>
link	<link *>	map	<applet *> </applet>
title	<title> * </title>	applet	<bgsound>
address	<address> * </address>	sound	<embed> * </embed>
blockquote	<blockquote> * </blockquote>	embed	<hr> *
div	*	horizontal rule	<img *>
heading x (where 0 < x < 7)	<hx> * </hx>	image	<marquee>* </marquee>
paragraph	*	marquee	<noscript> * </noscript>
bold	*	no script	<object> * </object>
big	<big> * </big>	object	<script> * </script>
cite	<cite> * </cite>	script	<spacer>
code	<code> * </code>	spacer	<caption> * </caption>
emphasise	*	caption	<col * >
font	 	column	<colgroup> * </colgroup>
italic	<I> * </I>	column group	<table> * </table>
strike	<s> * </s>	table	<tr> * </tr>
samp	<samp> * </samp>	row	<td> * </td>
small	*	data	<th> * </th>
span	*	header	<frame> *
strong	*	frame	<frameset> * </frameset>
sub	_*	frameset	<iframe> * </iframe>

super	[*]	floating frame	<noframes> * </noframes>
typewriter	<tt> * </tt>	no frames	<button> * </button>
underline	<u> * </u>	button	<fieldset> * </fieldset>
variable	<var> * </var>	field set	<form> * </form>
break	 	form	<input type=*>
center	*	input	<isindex> *
no break	<norb>	is index	<label> * </label>
pre	<pre> * </pre>	label	<legend> * </legend>
directory	<dir> * </dir>	legend	<optgroup> * </optgroup>
definition list	<dl> * </dl>	option group	<option> * </option>
definition item	<dd> * </dd>	option	<select> * </select>
definition term	<dt> * </dt>	select	<textarea> *</textarea>
unordered list	*	text area	
ordered list	*		

Appendix C − Recommended default keyword replacement for Java

This is the complete list of keyword replacements used during the HCI analysis.

Original Word	**Final text (* indicates final cursor position)**
if	if(*) { }
class	class * { };
print	System.out.println(*);
hash include	#include
hash define	#define

main	public static void main(String args[]) {
	*
	}

brackets	(*)

braces	{
	*
	}

libraries	import com.sun.java.swing.*; import java.util.*; import java.awt.*;

Appendix D – Version D User Manual

This manual takes you through all the stages of writing code using the CodeTalker. It assumes that you have a speech recognition package installed that you are comfortable using.

1. Starting up the CodeTalker

Double−click on the CodeTalker icon or bring it up via a command using the speech package. A standard text window will be displayed to you.

2. Creating a new file

When you start up the CodeTalker it will already have an empty document open. Before you can use all the functionality of the CodeTalker you need to save the file so that it knows what keyword replacements to make and what syntax to highlight.

To save the file say ‘Save as’. The words should be highlighted to indicate that you are saying a command. Now dictate the words that make up the name of the file e.g. ‘my file . java’. Once you have dictated the name say ‘tilda’. The text will be removed from the screen and the file saved.

Figure a: Saving a file

‘~’ is a terminating character that you use to tell the CodeTalker when you have finished dictating a command. To cancel a command at any point say ‘circumflex’. You can see both of these two letters and their uses in the prompt at the bottom right−hand corner of the editor as you dictate a command.

What languages does the CodeTalker currently support?

At the moment the CodeTalker supports programming in java and html. You can save a file with any extension but if it isn’t one recognised by the CodeTalker then it won’t perform replacement or syntax highlighting.

3. Write the code

3.1 Using auto−replacements

The CodeTalker automatically replaces certain words for you with other text as you speak. These words include print, replaced by System.out.println(); and if, replaced by 'if() { }'. A keyword is only replaced when you speak the word **after** the keyword. This may sound odd but means that if you dictate a word that contains a keyword e.g. printer, it won't get replaced accidentally.

Figure b: 'if' being replaced.

How can I add new replacements?

If you find there is some phrase that you are always saying you can create replacement for it to make programming faster.

1. Say 'create replacement ~'. A dialogue box will appear.
2. Say the word that you want to replace, then say ~ to continue
3. Now say the text that you want to replace the word with. This can be of any length and can contain new lines, punctuation etc. What you say here will be reproduced exactly.
4. Finally give the number of spaces back from the end of the text that you want the cursor to be after the replacement.

Figure c: Stages in creating a new replacement

The code talker will immediately start to replace the word that you have defined.

What about Syntax highlighting?

The CodeTalker automatically highlights syntax and you can add words that you want to be highlighted by saying ‘create keyword’. A dialogue box similar to that for creating a new replacement will appear, in which you should dictate the keyword.
You can stop the CodeTalker from highlighting syntax by saying ‘syntax highlighting off’.

Creating variable, class and function names

The CodeTalker attempts to automatically recognise when you are trying to dictate a class, variable or function name by examining the text around you. To let you know what it is doing it uses a traffic light system – words that it thinks will form a class name are coloured red, functions are amber and variables are green. In the bottom left−hand corner of the editor it also tells you what it thinks you are saying. Once it sees text that it recognises as a delimiter for a name (for example an opening bracket) it will concatenate the highlighted text to form a name automatically. You can force the CodeTalker to stop and create the name at any point by saying ‘tilda’.

Figure d: Screen shots showing a class, function and variable name being automatically recognised. The only words spoken were 'public class my class, braces, private void my function, open brackets, close brackets, braces, count equals my variable, semi−colon'.

There may be times when the CodeTalker doesn't realise that you want to say a name. To tell it simply say 'classname', 'variable' or 'function'. It will then start to read the text that you dictate as a name.

To stop the CodeTalker trying to interpret what you are saying as a name say 'circumflex'.

This sounds complicated, how long will it take to learn?

While this may sound complex it is actually simple to use. The best method is just to say the words that you want to appear and you'll find that most of the time the CodeTalker gets it right and does everything for you. If it doesn't just go back a step and tell it that you want to create a name or terminate the name yourself. This method was found to be quick to learn, mainly because there isn't really much too learn, just speak naturally and let the CodeTalker do the work.

Using the speech−directed interface

The CodeTalker allows you to access all the menu options by dictating commands into the editor window. Using the 'save as' and 'create' commands has already been discussed but there are several others.

All commands are highlighted so you can immediately see when the CodeTalker thinks you are trying to say a command. The format of the command and the characters to say to invoke the command (tilda) or cancel it (circumflex) are displayed at the bottom of the editor window.

What other commands are there?

Command	Action
open *filename*	Opens the requested file
File save	Saves the file
File new	Creates a new file
File close	Closes the file
Save as *filename*	Saves the file under the given name
View *integer* | *filename*	Switches focus to the tabbed pane position given or to the file with the given filename
Goto line *integer*	Jumps to line *integer*
Syntax highlighting on	Turns syntax colouring on
Syntax highlighting off	Turns syntax colouring off
Replace on	Turns word replacement on
Replace off	Turns word replacement off
Parse commands as I type	Enables parsing of commands as the user types
Wait for keyword	Only parses commands given after saying a specified keyword
Show line numbers	Displays line numbers
Hide line numbers	Hides the line numbers
Create keyword	Brings up the dialogue to allow new keyword creation
Create replacement	Brings up the dialogue to allow word replacement creation
Exit programme	Exits the programme, you will be prompted to save any files that have unsaved changes.

What if I want to say a word that the CodeTalker would interpret as a command?

By saying ‘wait for keyword’ you can make the CodeTalker wait for a keyword, set to ‘command’ by default before it parses for a command. This means you could now say ‘file’ or ‘create’ etc. and it would not enter command mode.

To return to making the CodeTalker parse everything as you speak say ‘command, parse commands as I type’.

What if I can’t remember a command?

Once you start to dictate a command the rest of the words that the CodeTalker expects to see are shown at the bottom of the window as a reminder.

Figure e: Details of the command are shown at the bottom of the editor

If you can't remember how to start the command say 'help me'. This will bring up a window showing all the commands available.

3. Compiling the programme

The CodeTalker does not provide it's own compiler. Most languages provide compilers that can be run from a DOS window or Unix shell and are easy to control by voice since the interface is text−based.

Before compiling the programme first save it by saying 'file, save'. If you need to know the full pathname of the file it is shown at the top of the editor window as can be seen in figure f.

Figure f: Screen shot to show the file name at the top of the window.

4. Exiting the CodeTalker

To exit the CodeTalker say 'exit programme'. The CodeTalker will prompt you to ask if you really want to exit. Say 'Yes' to close all windows and exit. If you have any open files with unsaved changes the CodeTalker will ask you if you want to save the changes before closing the file.

Figure g: Prompts displayed when exiting the programme

Appendix E – Cognitive Walkthrough

This was used to identify problems that users might have interacting with the system, either because it doesn't fit with their model of how a task should be performed or because it contains tasks that they have not previously performed. The interaction model of the CodeTalker was designed to be as similar to that described by the hierarchical task analysis as possible.

Need to consider :

- What impact will the interaction have upon the user ?
- What cognitive processes are required ?
- What learning problems may occur ?

Choose the tasks

Describe the users' initial goals
Work out the appropriate action to perform
Analyse the decision process

Opening a file

User Goal	***Action to perform***	***Compared To***
Open a file called test.java from directory c:\temp Let it know that I want to open a file Tell it the name of the file to open	Say 'Open' Say the filename	Select File menu Select Open option Enter filename
Possible problems :	None obvious	

Close file

User Goal	*Action to perform*	*Compared To*
Close the current file let it know that I want to close the file	File close	Select File menu Select Close option

Possible problems : None obvious

Change Windows

User Goal	*Action to perform*	*Compared To*
Change the window (assuming a tabbed pane view) let it know that I want view a different file	Window *number* Or View *filename*	Click on the tab for the document Or Select Window menu Select name of file to view

Possible problems : Window is used for numbers in the first instance, file names in the second.
Change to Tab *number* or View *filename* (or maybe View *filename / number* ?)

Change an option (use the syntax colouring option as an example)

User Goal	*Action to perform*	*Compared To*
Find the list of options Change the value of the relevant one (turn syntax colouring off)	*Syntax colour off* Say name of option Say setting to give it (syntax colour off)	Select 'Advanced' menu Click on 'Configuration' Click on the 'Syntax Highlighting' tab Check the 'Syntax Highlighting enabled' checkbox

Possible problems : Very different task structure but if all tasks options are given the same name as the base name when using UltraEdit it should be relatively intuitive for the user.
Need to make sure that this is the case (syntax colouring needs to become syntax highlighting)

Close the Application

User Goal	*Action to perform*	*Compared To*
Tell the application to exit	Exit program	Select 'File' menu Select 'Exit' option

Possible problems : Not obvious that you need to say program **but** 'exit' is a command used for programming so shouldn't really be used on it's own, the fact that you say it as a command is probably not a good idea. It would probably be better to group the exit command with the file command and have 'file exit'

Save the file for the first time

User Goal	*Action to perform*	*Compared To*
Save the file	File save	Select 'File' menu
let it know that I want to save the file		Select 'Save'
give the name to save the file under		Enter the name to save the file as

Possible problems : CodeTalker saves the file under it's current name which will be 'Edit*'. This may cause problems as you are not prompted to give a filename

7. **Save a file under a different name**

User Goal	*Action to perform*	*Compared To*
Save the file under a new name	Save file as	Select File menu
let it know that I want to save the file under a new name	*filename*	Select 'Save As' option
give the name to save the file as		Enter name to save file as

Possible problems : Different format again. All the other file commands come under 'File ….'. Would be better to have 'File save as *filename'*

8. Save the file after making changes

User Goal	*Action to perform*	*Compared To*
Save the file	File save	Select File menu
Tell it to save the file		Select 'Save' option

Possible problems : None obvious

Opening a new file

User Goal	*Action to perform*	*Compared To*
Open a new file Tell it to open a new file	File new	Select File menu Select 'New' option

Possible problems : None obvious

Writing Code

Include relevant libraries

User Goal	*Action to perform*	*Compared To*
Set up a series of '#include' statements that specify all the libraries to use Enter the '#include' keyword Enter the name of the library Repeat until all required libraries are listed	Hash include *filename* semi−colon new line Repeat until all required libraries are listed	Type '#include' Enter the name of the library to use Enter ';' to finish

Possible problems : None. The format for speaking is exactly the same as you would do when typing so

Define new class

User Goal	*Action to perform*	*Compared To*
Define a new class Indicate that I am creating a class Specify a name for the class Create an outline for the code that the class will contain to go in Define the details of the class	Class Classname *classname* Enter class details	Class *classname* Open brace Close brace Move cursor back inside braces Enter class details

Possible problems **:** None for 'class' − the editor puts all the punctuation in for you immediately after you've said 'class' so there would be no confusion with the user entering the punctuation and then finding that the editor also puts it in. Needing to say 'classname' before defining the name of the class may cause some confusion but this is a general method that needs to be learnt in order to use the VoiceCoder successfully so will just have to be put up with.

Define new function

User Goal	*Action to perform*	*Compared To*
Specify the function scope	Public | private | void	Public | private | void
Specify the return type	Classname *classname* OR void	*Classnam*e OR void
Specify a name for the function	Function *functionName*	*FunctionName classname variableName* : (Defines the parameters. Repeat as required)
Define any parameters that the function takes	Classname *classname*	
Create an outline for the code that the function will contain to go into	Variable *variableName* (Defines the parameters. Repeat as required)	Move cursor to after the brackets
Define the details of the class		Open brace
		Close brace
	Move cursor to after the brackets	Move cursor back inside braces
	Braces	Enter function details
	Enter function details	

Possible problems : Again the extra problem of saying 'variable' 'function' and 'classname' before the different names but, as previously mentioned, this will need doing no matter what if the voice coder is to be used correctly. The punctuation is added automatically and immediately so should not confuse the user. The word 'braces' is automatically replaced with a pair of braces so there is no inherent difference between the last steps (the user can still say 'open braces, new line, close braces' if they so wish.

Print data out on screen

User Goal	*Action to perform*	*Compared To*
Display data on the screen	Print	System.out.println
Use the System.out.println statement	Enter data to print	Open brackets
Pass it the values that you want to display	Move cursor to outside brackets	Enter data to print
		Close brackets
	Semi−colon	Semi−colon

Possible problems : The steps look different as the VoiceCoder requires you to move the cursor out of the brackets. This is because it has already put both the brackets in for you. In reality it is intuitive to move the cursor out of the brackets once you have finished (and the user could always change this setting)

Use 'if' statement

User Goal	*Action to perform*	*Compared To*
Define an if statement	If	Enter 'if' keyword
Indicate that I am defining an if statement	Define constraint	Open brackets
Define the clause for the statement	Move cursor to inside braces	Define constraint
Define the code to execute if the clause was true	Define code to execute if statement is true	Close brackets
		Open braces
		Define code to execute if statement is true
		Close braces

Possible problems : Again the steps look different due to the ordering of entering the punctuation (and the lack of this when using the voice coder). However as the punctuation insertion is immediate it is obvious to the user and only requires them to navigate around it. As mentioned the user could change the way that 'if' is replaced if they wanted to.

Use 'else if' statement

User Goal	*Action to perform*	*Compared To*
Define an else if statement	Else	Enter 'else if' keyword
Indicate that I am defining an else if statement	If	Open brackets
Define the clause for the statement	Define constraint	Define constraint
Define the code to execute if the clause was true	Move cursor to inside braces	Close brackets
		Open braces
	Define code to execute if statement is true	Define code to execute if statement is true
		Close braces

Possible problems : As for 'if'

Use 'else' statement

User Goal	*Action to perform*	*Compared To*
Define an else statement Indicate that I am defining an else statement Define the code to execute if the else clause is entered	Else Define the code to execute	Enter 'else' keyword Open braces Define code to execute if statement is true Close braces

Possible problems : Braces are added automatically. No obvious problems.

Use 'for' statement

User Goal	*Action to perform*	*Compared To*
Define a for statement Indicate that I am defining a for statement Define the base case Define the point at which looping should cease Define the increment to apply at each loop Define the code to be executed at each loop	For Variable name Equals Variable name or absolute value Variable name Comparison operator Variable name Increment value Move cursor inside braces Enter code to execute	Enter 'for' keyword Open brackets *VariableName* Equals Variable name / absolute value Semi−colon Variable name *Comparison* Variable name / absolute value Semi−colon *VariableName* ++, −− or +=, −= value Close brackets Open braces Enter code to execute Close braces

Possible problems : Although the two again look different the only real difference is putting in the punctuation. As the voice coder does this automatically after each word is spoken it is obvious to the user and the rest of the structure is the same.

Use 'while' statement

User Goal	*Action to perform*	*Compared To*
Define a while statement	While	Enter 'while' keyword
Indicate that I am defining a while statement	Variable	Define code to execute
Define the clause for the loop	Comparison	Open braces
Define the code to be executed at each loop	Variable or absolute value	Enter code to execute
		Close braces
	Move cursor inside brackets	Define finishing condition
		Open brackets
	Define code to execute	Define variable to use
		Define point at which looping should cease

Possible problems : Different to the normal method but still intuitive (as you are saying what you would usually type). Not something that should take long to get used to.

Define new variable

User Goal	*Action to perform*	*Compared To*
To define a new variable	Enter variable type	Enter variable type
Define the type that the variable should be	Variable *varname*	Enter variable name
Enter the name of the variable	*Or*	
	Create variable *varname*	
	Variable x	

Possible problems : Completely different from the usual method. However there is no way of replicating the typing method exactly using the voice. This is not a difficult set of instructions to learn

Assign a variable a value

User Goal	*Action to perform*	*Compared To*

Give a variable a certain value	Variable x	Varname
Define the variable to assign	Equals	Equals
Define the value to assign to it	Variable x or absolute value	Varname or absolute value

Possible problems : None, just need to learn how to access variables.

Use a function

User Goal	*Action to perform*	*Compared To*
Use a function	Function x	Function name
Define the name of the function to use	Variable x	Open brackets
Define the values of any parameters that the function needs	Comma	Define variables or absolute values to be passed
	(repeat until all required parameters are defined)	
	Move cursor outside brackets	Close brackets
	Semi−colon	Semi−colon

Possible problems : None