USING A “WIZARD OF OZ” STUDY TO INVESTIGATE ISSUES RELATED TO A SPOKEN LANGUAGE INTERFACE FOR PROGRAMMING

by

David E. Price

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2007

Copyright © David E. Price 2007

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

David E. Price

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

____________ ________________________

Co-Chair: Ellen Riloff

____________ ________________________

Co-Chair: Joseph L. Zachary

____________ ________________________

Cynthia A. Thompson

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of David E. Price in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Ellen Riloff Co-Chair, Supervisory Committee
Date	Joseph L. Zachary Co-Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Grappling with the syntax of a programming language can be frustrating for programmers because it distracts from the abstract task of creating a correct program. Novice programmers often struggle because they are forced to learn syntactic and general programming skills simultaneously. Persons with vision loss and persons with mobility impairments have problems creating and debugging syntactically detailed programs. Even experienced programmers may be hampered by the need to learn the syntax of a new programming language.

A spoken language interface for programming provides one solution to these syntax-related problems. In this thesis, I propose one possible design for such a system and conduct a study to investigate several issues related to how people might use it. My goal is to shed light on both the promise and challenges behind future work in this area.

First, I developed a prototype natural language interface for Java programming, called NaturalJava. In NaturalJava, the user types English sentences and describes, step by step, the desired program. In response to each of these sentences, the interface produces syntactically correct source code and displays it to the user. NaturalJava's architecture comprises three components: (1) Sundance, a natural language processing system; (2) PRISM, a knowledge-based module that interprets user requests; and (3) TreeFace, an abstract syntax tree manager that maintains the evolving source code. While NaturalJava has a number of limitations, this proof-of-concept prototype successfully demonstrates how such a system might be developed using current technology.

Second, I conducted a Wizard of Oz study to answer several questions about how people would ultimately use a fully functional version of a spoken language programming interface similar to NaturalJava. I investigated three specific issues:

the complexity of the actions requested by the users, the extent to which people use a similar vocabulary while programming, and the impact of artifacts of speech, such as disfluencies, on the ability of the interface to function. My results suggest that: (1) novice programmers request relatively simple actions from the interface, (2) they utilize similar vocabularies while making these requests, (3) disfluencies will pose significant problems for such an interface, but information extraction technology is relatively resilient in the presence of these disfluencies, extracting $\approx 90\%$ of the data successfully.

To my parents

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **ix**

LIST OF TABLES . . . . **xi**

ACKNOWLEDGEMENTS . . . . **xiii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Chapter Summary 6

2. RELATED WORK **8**

2.1 State of the Art for Access Technology 8
2.1.1 Access for Persons with Vision Loss 9
2.1.2 Access for Persons with Mobility Impairments 10
2.2 Syntax-free Programming Systems 11
2.2.1 Automatic Programming 11
2.2.2 Structure Editors 14
2.2.3 Natural Language-Based Programming 15
2.2.3.1 NLC 15
2.2.3.2 MOON 17
2.2.3.3 TUJA 19
2.2.3.4 Metafor 20
2.2.3.5 VoiceCode 20
2.3 Wizard of Oz Studies 22
2.4 Speech Disfluencies 22
2.5 Chapter Summary 25

3. THE NATURALJAVA PROTOTYPE **27**

3.1 The Architecture of the Prototype 28
3.1.1 Sundance 32
3.1.2 PRISM 36
3.1.3 TreeFace 39
3.2 The Limitations of the Prototype 40
3.3 Converting to Other Programming Languages 40
3.4 Chapter Summary 41

4. THE DESIGN OF A WIZARD OF OZ STUDY **43**
4.1 Subjects 44
4.2 Wizards 46
4.3 Subject Feedback 47
4.4 Chapter Summary 48

5. WIZARD OF OZ STUDY RESULTS **50**
5.1 Data Collection 51
5.2 Command Type Results 53
5.3 Vocabulary Results 59
5.4 Disfluencies 66
5.4.1 Data Preparation for Disfluencies 66
5.4.2 Disfluencies in Programming Requests 67
5.4.3 Disfluency Impact on NLP 70
5.5 Informal Feedback 77
5.6 Chapter Summary 80

6. CONCLUSIONS **82**

REFERENCES **87**

LIST OF FIGURES

2.1 The add-to complement structure used in NLC. 16

3.1 The NaturalJava user interface. The largest of the three text areas contains the evolving Java source code. The text area at the bottom of the screen holds error messages and requests for more information from the user. The text area along the right side of the window provides other data requested by the user, such as the variables in scope or the methods associated with a class. 28

3.2 Input from User 1's NaturalJava session to create a priority queue class. Note that NaturalJava does capitalization correction on class, method, and variable names. 29

3.3 Input from User 2's NaturalJava session to create a priority queue class. Note the differences between these instructions and those in Figure 3.2, which both create the same priority queue class. 30

3.4 Output from a NaturalJava session. The input sentences shown in Figures 3.2 and 3.3 result in the Java source code seen above. 31

3.5 Architecture of the NaturalJava prototype. 33

3.6 An example of a case frame in Sundance. This case frame is instantiated when the word "iterates" is used as an active voice verb construction. 36

3.7 The instantiated case frames produced by Sundance given the sentence "Create a for loop which iterates from 1 to 10." 37

5.1 The growth of a shared vocabulary. The line shows the average overlap in vocabulary for each of the subjects when compared against a vocabulary generated by a given number of other subjects (on the x-axis). Error bars show the range in overlap for the individual subjects. 63

5.2 The growth of a common vocabulary when utterances containing source code comments are removed. The line shows the average overlap in vocabulary for each of the subjects when compared against a vocabulary generated by a given number of other subjects. Error bars show the range in overlap for the individual subjects. 65

5.3 The growth of coverage in word instances. The line shows the average coverage of word instances for each of the subjects when compared against a vocabulary generated by a given number of other subjects. Error bars show the range of coverage for the individual subjects. 67

5.4 An example of an instantiated Sundance case frame triggered by the verb “create”. 72

5.5 A pair of case frames that satisfy the head noun match criteria. The original utterance to produce case frame A is “create a um public class”. 73

5.6 A pair of case frames that satisfy the slot name match criteria. The original utterance to produce case frame A is “create a um let’s see public class”. 74

LIST OF TABLES

3.1 Summary of the case frame types used by Sundance in NaturalJava. . 35

5.1 Example entry in collated session data file. The manual transcripts include information about pauses in the subject's speech. In this case, *[mp]* represents a medium-length pause (3–10 seconds). 52

5.2 The command type tags applied to the Wizard of Oz data. 53

5.3 Statistics on the data set collected during the Wizard of Oz study. . . . 54

5.4 Distribution of command type usage. 55

5.5 Frequency of the number of different command types per user request. 56

5.6 The 15 most common command tag clusters, without regard to tag order. The number of times each cluster occurs, and this number as a percentage of all commands in the corpus, are given. These 15 command tag clusters account for 75% of all command tag clusters. . . 57

5.7 Number of tags present within a request as a percentage of a given command tag's usage. 59

5.8 Percentage of each subject's vocabulary that overlaps with a vocabulary established by the other seven subjects. 61

5.9 The growth of vocabulary coverage as more subjects are added. The number in the left column represents the number of other subjects used to build the base vocabulary. 62

5.10 Statistics for the data set for C++ subjects. The utterances excluded from the data set contain source code comments. 64

5.11 The growth of vocabulary coverage as more subjects are added. The number in the left column represents the number of other subjects used to form the base vocabulary. All utterances containing source code comments were removed before comparison. 64

5.12 The growth of word instance coverage as more subjects are added. The number in the left column represents the number of other subjects used to build the base vocabulary. 66

5.13 Rates of disfluencies across subjects. The right-hand column shows the number of filled disfluencies that occur per subject request. 68

5.14 The number of disfluencies, listed by type, contained within a subject's requests. Column headings are: FP = Filled Pauses; FS = False Starts; RPT = Repetitions; RPR = Repairs. 69

5.15 Distribution of filled disfluencies across types. 69

5.16 Statistics on the case frames generated by Sundance and the numbers of case frames that match for both the original utterance and the effective utterance. Numbers for case frame matching are cumulative. 75

ACKNOWLEDGEMENTS

I would like to thank my advisors, Ellen and Joe, for allowing me to pursue this unusual topic. I would also like to thank my parents for all their support. Many thanks to Ben Newton and Dana Dahlstrom for their work developing the testing infrastructure and being “wizards”. Thanks also to Hung Huynh for being a wizard, Ryan Pratt for transcribing user requests, and Henry Longmore for transcribing and tagging the user requests. Thanks to Betty Mohler, Bill Phillips, and Cindi Thompson for testing early versions of the Wizard of Oz infrastructure.

This research would not have been possible without the many students who helped to test the testing infrastructure or who were subjects in this research. The work described here was supported by the NSF under grants DUE-0088811, IIS-9704240, IRI-9509820, and IIS-0090100.

CHAPTER 1

INTRODUCTION

This research was motivated by my experiences taking introductory computer science classes. As a blind student sitting in class, without access to the information being written on the board, unable to read the textbook, and with a two-week turnaround to obtain class notes in a form I can access, I became hopelessly lost in the syntax of a new programming language. I also discovered that I had a very difficult time finding and correcting syntax errors in my code. One possible solution to this problem for someone who is blind is to design a syntax free method of programming. This would allow the user to move to a higher level of abstraction while programming—they can deal with the concepts of programming without worrying about the details of syntax.

This method of programming could be very useful for many other people as well. Novice programmers often struggle because they are forced to learn syntactic and general programming skills simultaneously. People suffering from mobility impairments, including those caused by repetitive strain injuries, have difficulties entering syntactically complex code from a keyboard. Even experienced programmers may be hampered by the need to learn the syntax of a new programming language.

After giving it some thought, I decided that using spoken English sentences would be the best syntax-free input language. I made this decision for two reasons. First, people will not need to be trained to use an intermediate language before they can begin to work. A person can just describe, in English phrases, the desired programming constructs, and create syntactically correct source code. Second, tools for processing speech and extracting information from English sentences already exist, and can be used to simplify the process of building any future interface.

The major drawback to using natural English as the input language is the inherent ambiguity of the language. Words in English do not have unique and concise meanings, and the meaning is often strongly dependent upon the context of its occurrence. The word "add" exemplifies the ambiguous nature of English. For example, in the phrase "add x to y", add takes on the meaning of mathematical addition. Alternatively, in the phrase "add a method to this class", add takes on the meaning of incorporating an element into a greater whole. Finally, in the phrase "create a method called add", it represents the name of a method.

This is in direct contrast to the syntax of a programming language, which contains words and symbols with concrete meanings which are only occasionally dependent on context. For example, the symbol '*' can have different meanings based on context in the C programming language. When placed between two variables that store numbers, it represents multiplication. On the other hand, when placed before a variable that points to an address in memory, the combination of '*' and the variable name refers to the value stored at the memory address, and not to the memory address itself. However, there is a crucial difference between natural languages and programming languages when it comes to context. A programming language has rigidly defined rules for resolving the meaning of symbols based on their context, while natural languages do not.

Therefore, to test whether a natural language interface for programming was possible, I implemented a prototype, called *NaturalJava*, that takes natural language sentences and sentence fragments from the keyboard and converts the requests into syntactically correct Java source code [17].

The prototype works within a small subset of the Java language and, more significantly, within the small subset of the English language that was sufficient for me to create small Java programs. The interface exploits three subsystems. The Sundance natural language processing system accepts English sentences as input and uses information extraction techniques to generate case frames representing program construction and editing directives [22]. A knowledge-based case frame interpreter, PRISM, uses a manually created decision tree to infer program mod-

ification operations from the case frames. A Java abstract syntax tree manager, TreeFace, provides the interface that PRISM uses to build and navigate the tree representation of an evolving Java program [17].

This prototype presents one proposed architecture for how a natural language interface for programming might be designed. But many technical challenges and questions must still be addressed in order to create a fully functional, robust, and accurate spoken language interface for programming. The goal of this thesis is to investigate three specific issues that would impact the development of spoken language interfaces for programming. I will investigate these issues in the context of a group of novice programmers taking a single introductory programming class. These issues are represented in the following research questions:

- *What types of commands do the people use and how do they use them?*
- *Do the people share a common vocabulary or do the words they use differ substantially from person to person?*
- *What types of disfluencies are used by the people, how frequently do these disfluencies occur, and how do they impact the underlying natural language processing (NLP) system that extracts information from the user requests?*

To answer these questions, I need to study how these students would use a spoken language interface for programming. This leads to the problem of studying how people use an interface before it is built. Unfortunately, the prototype cannot be used for this task. The prototype accepts input typed from the keyboard. Even if I added a speech recognition system in place of the keyboard, the prototype could not be used because its coverage of the vocabulary used for programming is limited, and the prototype cannot carry out all of the different editing operations that might be needed. In short, the prototype cannot carry out arbitrary requests from an arbitrary user. Expanding the prototype to have a fuller vocabulary and more editing features is also problematic because I do not know the vocabulary or features that an arbitrary user would be likely to use. Therefore, I needed a

way to have people use a spoken language interface for programming without first developing the system for them to use. This is the basic premise of a *Wizard of Oz* study.

A Wizard of Oz study allows researchers to evaluate the possible design of a new system before actually implementing it, and can be crucial in uncovering design flaws and identifying research issues before the system is developed. People interact differently with computer programs than they do with other people [16]. If a researcher wants to learn how people will use a program, it is necessary to convince them that they actually are using that program.

In a Wizard of Oz study, the test subject sits in front of a computer displaying the interface to "the new program". An expert, called "the wizard", sits at a second computer located in a different room. The two computers are connected over a network, allowing the wizard to watch and/or listen to every action taken by the subject. The subject, however, is unaware of the presence of the wizard, and believes that he or she is working with an existing program. The expert listens and/or watches every input from the test subject, determines what the subject is trying to do, and then places the appropriate response from the imaginary program into the interface on the subject's computer. In this way, the subject, believing that the computer program exists, demonstrates how they would use the program. The subject's actions and the wizard's responses can be used to aid in the future development of the program being simulated.

For this thesis, I carried out a Wizard of Oz study. Two undergraduate research assistants and I developed the hardware and software infrastructure needed to carry out the study. After testing this infrastructure, I conducted the study using eight novice programmers from an introductory C++ programming class. The students who volunteered to take part in the study spent two hours each week working on their class assignments using the simulated programming interface. The subjects uttered 8117 requests to the interface during 67 sessions. Additionally, each subject completed a questionnaire describing their successes and problems at the end of each session.

The results for the research questions I investigated for this study are:

- *What types of commands do the people use and how do they use them?*
  Command types indicate the types of actions requested by the subject (such as navigation, edit, declaration, etc.). Almost all of the requests made by subjects were simple in nature, containing only one or two command types within the request. Overall, subjects predominantly used a small subset of the possible command types available to them. Examining the groups of command types used within the individual requests shows that a relatively small collection of command type groups comprise most of their requests.

- *Do the people share a common vocabulary or do the words they use differ substantially from person to person?*
  Subjects used a wide variety of words to express their requests to the interface. However, comparing each subjects vocabulary against the collective vocabulary of the other subjects shows that the collective vocabulary covers, on average, almost 80% of the individual subject's vocabulary. These results suggest a strong commonality of vocabulary across subjects.

- *What types of disfluencies are used by the people, how frequently do these disfluencies occur, and how do they impact the underlying natural language processing (NLP) system that extracts information from the user requests?*
  Disfluent speech is the portion of an utterance that must be removed to achieve the speaker's intended utterance. Disfluent speech includes filled pauses, false starts, repetitions, and repairs. In the data collected during the study, disfluencies occur in about 50% of the utterances. Filled pauses are the most common, but the proportions of the various disfluencies vary among the subjects. Surprisingly, the disfluencies found in these data have a relatively small impact on the underlying natural language processing system that supports NaturalJava.

These results are limited in a number of ways. First, all of the subjects were novice programmers. Second, all of the subjects were students in a single introduc-

tory programming class. The instructor may have influenced how they described programming constructs through his use of language during class. Third, this study covers a small number of subjects performing a relatively small number of tasks. As a result, the data collected during this study may not represent a general population.

In the next chapter, I discuss the current state-of-the-art in access technology and previous research in syntax-free methods of programming, Wizard of Oz studies, and spoken language disfluencies. Chapter 3 describes the NaturalJava prototype that I developed. In Chapter 4, I describe the Wizard of Oz study that I have completed. Chapter 5 discusses the results from analyzing the data collected during the Wizard of Oz study. Finally, I present my conclusions drawn from this research.

1.1 Chapter Summary

The problems posed by programming language syntax motivated this research. Syntax-related problems impact many different groups of users, including novice programmers, persons with vision and mobility impairments, and advanced programmers using a new programming language. One potential solution to these problems is a spoken language interface for programming.

I implemented a prototype natural language interface for programming to determine if it was possible to create program source code using natural language. The prototype illustrated how such a system might be designed, but I realized that extending it to be a complete system raised questions about how users would interact with such a system. Consequently, I conducted a Wizard of Oz study to collect data on how a group of novice programmers would interact with such an interface. I then used these data to investigate three aspects of a spoken language interface for programming. The results of this study show that the subjects requested simple actions of the interface and used a similar vocabulary. Speech disfluencies were common but the underlying natural language processing system performed surprisingly well despite these disfluencies. These results are

limited by several aspects of the study and, as a result, may not generalize to other populations.

CHAPTER 2

RELATED WORK

In this chapter, I will describe previous research relevant to my work. First, I will discuss the present state of the art in access technology—the technology that enables persons with disabilities to use computers. Next, I will describe previous syntax-free methods of programming. These previous efforts can be divided into three categories: automatic programming, structure editors, and natural language interfaces for programming. In Section 2.3, I will describe some previous work that used Wizard of Oz studies. Finally, human speech often contains linguistic artifacts that must be removed to achieve the speaker's intended utterance. These artifacts are called speech disfluencies. A spoken language interface for programming would have to deal with speech disfluencies. Therefore, I will explain how previous research has characterized disfluencies in the final section of this chapter.

2.1 State of the Art for Access Technology

Accessible computer technology is a field still in its infancy. Almost all of the access technology currently developed is designed to allow persons with disabilities to work with the mainstream applications commonly available today. These interfaces are developed by small- to mid-sized software companies, almost all of whom specialize in adaptive technology.

Most mainstream software developers do not consider access issues during the development process. An excellent example of this can be found in software that automatically generates pages for the world wide web. Guidelines for the production of accessible web content have been developed by the World Wide Web Consortium [4]. Unfortunately, few developers incorporate these guidelines into their software, so much of the content generated for the world wide web is difficult for the blind

to access. A second example of this is associated with the Java programming language. Sun Microsystems has built accessibility features into the Java API. Unfortunately, few Java developers incorporate these features into their software, nor is it commonly taught to students learning Java.

While a few major software suppliers have begun to address access issues, their efforts have been less than successful. For instance, Microsoft has added "Accessibility Options" to the Windows operating system. Some of the simpler options are successful, but most are not. For the vision-impaired, the high contrast display settings are successful, but the screen magnification and cursor/mouse enhancements are not useful. Most importantly, these enhancements are of no use to most blind computer users, who are unable to see the display. Microsoft has introduced the Active Accessibility API (MSAA), which makes information from programs available to access technology software. This API enables a few programs, such as Microsoft Internet Explorer and Adobe Reader, to provide high-quality access to mainstream applications for the blind. Unfortunately, many software developers do not implement this API for their applications. For instance, Microsoft does not utilize this API for many of their applications, such as the Office suite or VisualStudio.

2.1.1 Access for Persons with Vision Loss

The bulk of the blind community uses screen reading software to access a computer. These programs keep track of the user's actions, such as typing or moving the cursor through a document, and the computer's actions, such as creating new windows or printing status messages, and sends the appropriate information to an output device. One type of output device displays braille. Unfortunately, braille displays generally cost about $10,000, so are too expensive for many persons with vision impairments to purchase. Additionally, only 10% of persons with vision loss in the United States can read and write in braille. This figure is reduced further because it is difficult for people who lost their sight as adults to achieve a high level of braille proficiency. For them, reading braille is slow and cumbersome. Therefore,

in the United States, most blind computer users use a speech synthesizer for their screen reader output.

Speech output poses several challenges to the blind programmer. First, typing mistakes, especially capitalization errors, are difficult to catch, because the misspelled word may sound identical to the correctly spelled word. Second, finding errors in syntax is difficult. For example, the sighted programmer has visual cues, such as the level of indentation used to aid in matching braces. Visual information of this kind is difficult to convey through speech. Third, scanning through the source code is time-consuming. A screen reader does not have the ability of the human eye to pick out individual words from the code to get a sense of the position within the file. The screen reader must read across the line starting from the left edge. Thus, the blind user must read many lines to get a sense of the current position. This strongly affects one's ability to quickly navigate through the source code.

2.1.2 Access for Persons with Mobility Impairments

There are many different types of motor impairments. This results in a wide variety of adaptive strategies and devices, depending on the nature of the impairment. For those users with some mobility of the hands, there are specially designed keyboards, mice, track balls, and joy sticks. The keyboards have fewer keys, and rely on "key-chords" to provide access to all of the functions of a standard keyboard. Pointing devices have larger buttons, that are often color-coded, and have been modified to permit easier use. For those persons who do not have use of their hands, other types of pointing devices are used with on-screen keyboards. These pointing devices may be activated by feet, eye movement, mouth, or infrared pointers mounted on headsets. To type, the user must point at a "key" on the screen and then activate a "mouse-click." There are computer applications, such as word prediction software, that can increase the productivity of these users. Finally, there are speech driven interfaces that can be used as input to a computer.

Unfortunately, none of these computer interfaces are well-designed for programming tasks. Use of the keyboards and pointing devices is slow. Productivity

enhancement applications, such as word prediction software, can help somewhat with standard English words, but consider the problems associated with variable, method, and class names, such as numberOfElements or elementAtIndex. These names will not appear in standard dictionaries. The user will either have to type out every name in full or have to add every such name to the dictionary. Speech recognition packages allow a faster method of entering the source code. However, this method is also cumbersome, since every aspect of the syntax (every parenthesis, every semicolon, every comma, etc.) must be dictated. Variable, method, and class names will also have to be added to the speech dictionary.

2.2 Syntax-free Programming Systems

NaturalJava is not the first attempt to produce a syntax-free method of programming. The previous work on syntax-free programming spans three fields: automatic programming, structure editors, and natural language interfaces for programming. I will briefly describe the work that has occurred in each of these fields in the sections below.

2.2.1 Automatic Programming

The goal for automatic programming has been a moving target over the years. Initially, compilers and assemblers were considered automatic programming, since they took a higher level language for programming and converted it into machine language. Since that time, the goal for automatic programming has evolved to become the process of generating computer programs from an end-user's nontechnical description. The discussion that follows is based on the survey found in Rich and Waters [18].

To achieve the current goals, an automatic programming system would require three features:

1. The system is end-user oriented. In other words, the system can communicate in a language the user understands.

2. The system is general-purpose. It works equally well in all domains.

3. The system is fully automated. Once given a set of requirements, it requires no human assistance to generate an efficient program.

Unfortunately, current technology cannot satisfy these requirements.

For an automatic programming system to be end-user oriented, it must have knowledge of the application domain. The user's specification for the program would be described using the terminology of the application domain. This specification must be converted into a specification in the programming domain. To perform this conversion, the automatic programming system must have a knowledge of the terminology used within the application domain comparable to the knowledge of the end-user. For example, consider the following user specification:

> *"The function EvalOctal is a recognizer that determines whether or not a given string contains an octal number optionally surrounded by blanks. If this is the case, the decimal value of the number is returned; otherwise, -1 is returned."*

To process this specification, the automatic programming system must understand many terms, including recognizer, octal number, and decimal number. The system must also know the procedure for converting an octal number to a decimal number. While the required knowledge can be assembled for very narrow domains, broadening the domain knowledge has proven difficult.

This need for domain knowledge also creates a problem in meeting the second requirement for these systems. In order to be general purpose, an automatic programming system must have domain knowledge for every possible domain. This requirement is unrealistic at the present time, since the field of artificial intelligence has not developed the capability to store and access such a knowledge base.

Finally, program specifications are rarely complete. In the relatively simple *EvalOctal* example given above, several aspects of the desired program were not fully specified. For instance, would the program need to process negative octal numbers? A human programmer would make a good faith effort to complete the program, making reasonable assumptions when necessary. In the *EvalOctal*

example, the programmer would assume that only positive octal numbers must be processed; otherwise, the error condition (-1) could also represent a valid return value for the program. An automatic programming system would need to be able to make inferences such as this in order to be effective.

However, when many possible solutions can be used to fill in gaps within the specification, the human programmer would enter a dialog with the end-user to constrain the specification. This is a common situation, since end-users often do not know exactly what they desire when they first specify a program. In this situation, the programmer enters a dialog with the end-user, refining the specification until it is complete. An automatic programming system would need to engage in this dialog, specifying its assumptions as well as forming questions to resolve problems with the specification. Thus, achieving a fully automatic system is problematic.

Given the difficulties outlined above for each of the three desired features in an automatic programming system, three approaches to solving the problem are being attempted. Each of these approaches sacrifices one of the desired features in an attempt to achieve the other two. These approaches are:

1. *Bottom-up:* End-user orientation is sacrificed by this approach. It starts from the programmer's level and pushes the level of automation upwards. This approach limits the amount of domain knowledge needed by the system because specifications remain much closer to the programmer's domain than the end-user's domain. The high level languages used today, such as Java and C++, represented the first step on this path. Current research focuses on very high level languages, such as SETL [8].

2. *Very Narrow Domain:* This approach sacrifices the system's ability to be general-purpose. Since the knowledge needed to permit automatic programming can be developed for very narrow domains, this approach develops fully functional automatic programming systems that work within these narrow domains. Current research for this approach focuses on broadening the knowledge domains.

3. *Assistant:* This approach reduces the amount of automation in an effort to assist the user. This approach focuses on developing tools and interfaces to assist in different levels of program development. Current work focuses on improving the levels of assistance provided by the tools and integrating these tools into more helpful interfaces.

At first glance, it may appear that NaturalJava (see Chapter 3) fits within the "bottom-up" approach to automatic programming. However, the NaturalJava system is not attempting to perform automatic programming. While the interface is designed to accept natural English sentences and sentence fragments, the content of the input is not a high-level specification of the program to be created. Input to this program is a step-by-step description of the contents of the source code.

2.2.2 Structure Editors

Structure editors, also known as syntax-directed editors, have been used as an educational tool for teaching introductory computer programming [15]. Unlike text editors, structure editors do not allow the user to create a syntactically incorrect program. Structure editors use an abstract syntax tree as the internal representation of the program. As a result, each modification to the evolving source code being written by the student is governed by the grammar of the programming language, resulting in error-free syntax for the student.

The user modifies the evolving program by choosing options from a menu and then filling in the required fields in a form. The menu options available at any given time are limited to those modifications permitted by the grammar of the programming language at the current node within the abstract syntax tree. These restrictions on the possible manipulations of the abstract syntax tree often make it difficult to make large-scale changes to the evolving source code because intermediate steps result in illegal syntax. These types of problems have been circumvented by allowing the user to work outside of the structure editor for short periods of time.

Structure editors do not represent a viable alternative for programming for the vision-impaired. They require the user to navigate through sequences of menus and forms for each new construct. Although it would guarantee correct syntax, it would be cumbersome and, at times, confusing for someone with visual impairments.

2.2.3 Natural Language-Based Programming

Two natural language interfaces for programming predate NaturalJava, while three interfaces were developed after the NaturalJava project began. The two systems that predate NaturalJava are NLC [1] and Moon [26] while the other three are called TUJA [11], Metafor [13], and VoiceCode [6]. Four of these interfaces (NLC, Moon, TUJA, and Metafor) take sentences from the keyboard as their input, while VoiceCode uses spoken language input. I will briefly describe these systems in the sections below.

2.2.3.1 NLC

NLC [1] is a natural language interface that allows users to manipulate tables and matrices by typing English sentences. Four modules comprise the system: the scanner, the parser, the semantics analyzer, and the interpreter. Each of these modules run in sequence, and there is little interaction between the different modules. These modules perform the following functions:

- *The Scanner*

  The scanner divides the input string up into the individual words, called tokens. If the word represented by a token is located in the dictionary, the associated attributes are attached to the token. The authors do not detail what happens if the word is not found in the dictionary.

- *The Parser*

  The parser is used to determine the structure of the input sentence, given the information provided by the scanner. The authors state that most inputs in this domain follow the top-level structure of an imperative verb followed by its operands. They use an augmented transition network to generate a

syntactic structure for the sentence, and the noun phrases can be placed into the appropriate complement structure for the given verb. These noun phrases represent the operands for the matrix manipulations specified by the imperative verb. For example, consider the input sentence:

> *"Add the first positive entry that was doubled in row 3 to the second to last row."*

After parsing, the resulting noun phrases would be placed into the add-to complement structure, which might look something like Figure 2.1. The authors do not describe examples of sentences that do not contain imperative verbs.

- *The Semantic Analyzer*

  The primary task for the semantic analyzer is to determine, from the data contained within a parse tree, the real world "objects" being specified by the English input sentence. For example, the semantic analysis for the noun phrase

  > *"the last positive entry in row 4"*

  would locate the column in row 4 in which that value occurred and replace the noun phrases with that matrix address. For example, if the last positive value in row 4 is found in column 5, the phrase above would be replaced by the matrix coordinates (4,5).

```
add-to { np: ''the first positive entry that was doubled in row 3''
   pp_to: ''the second to last row'' }
```

Figure 2.1. The add-to complement structure used in NLC.

This module also performs coreference resolution to locate the appropriate addresses for the requested calculations. For example, given the request "Add 5 to the row that was doubled", the semantic analyzer must review the previous user requests and their semantic analyses to determine which row was most recently "doubled". It can then substitute the appropriate row address for the phrase "the row that was doubled".

- *The Interpreter*

  The interpreter takes the imperative verbs and the addresses determined by the semantic analyzer and carries out the requested modifications to the matrix. This is done in two steps. First, the verb is mapped to the associated operation that will be carried out on the matrix. For instance, "double", "triple", "negate", "square", and "cube" are all considered to be special cases of the "multiply" operation. Next, having ascertained the operands for the calculation in the semantic analyzer, the calculations can now take place and the display updated to show the changes.

The NLC system was tested by twenty-three undergraduates taking their first programming class. They were asked to solve one of two test problems. The assigned problems were solved by 74% of the students within the two-hour time limit. NLC processed 81% of their requests immediately, and most of the remaining requests were rephrased and successfully processed.

2.2.3.2 MOON

MOON [26] is a natural language-based programming system that is designed to obviate the need for learning syntax. Their goal in developing this system was to *"enable the comment to act as the code, making the code readable, maintainable, and reusable."* This should allow for non-technical people to develop new data structures and methods without requiring an in-depth knowledge of syntax and programming language structure. While this goal sounds similar to automatic programming, this programming language falls short of that mark. It requires the

user to write step-by-step instructions for program execution. It also requires the user to have a strong grasp of the concepts of object-oriented program design.

Moon differs from NaturalJava in a significant way. Moon does not transform its input into an existing programming language. It is a programming language in its own right, and must run within an interpreter. In contrast, NaturalJava transforms its input into the constructs of the Java Programming Language.

Moon is an object-oriented language that implements polymorphism and inheritance. In this language, all objects are indicated by an initial capital letter. The interpreter scans through the sentence, identifies the objects based on this capitalization, and pushes the objects onto the stack. It then examines the remaining words of the sentence to identify the relevant command being invoked. For example, given the input, "copy Number from Origin to Destination.", the interpreter will place Number, Origin, and Destination on the stack, find 'copy from to' in the remaining words, and invoke the Number's copy method, passing Origin and Destination as arguments. This allows the Moon interpreter to use method names that are more than one word or phrase in length.

The Moon interpreter also utilizes a fault-tolerance system to improve ease of use. It has three principle tasks:

- Suggesting correct spellings (e.g., suggesting "argument" when "agrument" is typed)

- Rearranging sentences so that the arguments are pushed onto the stack in the correct order (e.g., "copy Number to Destination from Origin becomes "copy Number from Origin to Destination")

- Suggesting the correct method name if the sentence does not contain one because of missing words or incorrect prepositions. For instance, if the user wrote "index at position Pos on stack CheckStack", Moon would suggest "index of argument at position Pos on stack CheckStack" because the most similarly defined method included the phrase "of argument".

Moon only approximates natural language understanding. It assumes that the predicate (i.e., the word containing the most important information) is the first word in the sentence that is not an object or a preposition. While this allows some tortuous sentence structures (e.g., "from Origin copy Number to Destination"), it allows the interpreter to quickly ascertain the crucial words in the sentence without performing much processing of the natural language used as input.

Moon has been used to implement a prototype for a digital workplace supporting a care process for nursing. Its aim is to allow the staff of a nursing facility to create and modify process checklists and notification systems for patients based on changing needs. It was ready for testing in 1997.

2.2.3.3 TUJA

TUJA [11] is a project to develop an interface that takes Turkish sentences from the keyboard as input and produce Java source code as its output. In 2003, TUJA was capable of creating a Java class skeleton (i.e., class, method, and data member declarations).

TUJA uses an augmented transition network for parsing and semantic analysis of the natural language input. TUJA classifies input sentences into four categories: class declarations, member declarations, method declarations, and relationship declarations (i.e., inheritance and polymorphism). In general, TUJA assumes that nouns refer to classes, interfaces or members, while verbs refer to methods. TUJA classifies verbs into four categories based on whether the action requires parameters as input and/or returns values.

Knowledge derived from the ATN is stored in schemata. There are three different types of schemata: method schema, member schema, and class schema. TUJA uses a Prolog relational data base to store these schemata, relying on HASA relationships for composing classes and ISA relationships for building class hierarchies. The authors define a HASA to be the inclusion relation, indicating that other objects are elements of a larger object. For example, a class has a method.

TUJA does not show the evolving source code to the user. However, at any time the user can request that Java source code be written to a file. No user testing

has been performed using TUJA, to date.

2.2.3.4 Metafor

Metafor [13] employs the idea that programming is similar to storytelling. It builds a “scaffolding code” (which may or may not be executable) to demonstrate the framework of the solution. It utilizes the notion that details about procedures can be inferred from linguistic structures. This process, called programmatic semantics, infers data structures from noun phrases, functions from verbs, and properties from adjectives. Some linguistic structures indicate the presence of conditionals, loops and recursive structures.

Metafor behaves similar to a prose outlining tool. As a programmer types more and more of the “story” of the code into the interface, Metafor updates its representation of the program with each sentence. The resulting code is displayed in Python.

The authors conducted a preliminary user study with 13 participants, six of whom were novice programmers and seven who were intermediate programmers. The purpose of the task was to evaluate Metafor’s usefulness as a brainstorming tool for writing source code. The subjects implemented, in story form, the actions of the characters in the video game Pacman, first on paper and then using Metafor. The subjects self-assessed the usefulness of brainstorming for producing a solution to the problem both on paper and using Metafor. All subjects preferred brainstorming using Metafor over paper, and preferred paper over not brainstorming. Novice programmers were more enthusiastic about the benefits of Metafor than were intermediate programmers.

2.2.3.5 VoiceCode

VoiceCode [6] represents the first spoken language interface for programming. It was developed to assist programmers suffering from repetitive strain injuries. It accepts spoken instructions using a defined syntax and produces source code in one of three programming languages (ex, Python, or C++). This interface also allows the programmer to navigate and modify the source code by uttering a continuous

stream of voice commands.

VoiceCode uses templates to generate common programming constructs, such as loops and class declarations. Several specific keywords are associated with each of these templates to allow the user some flexibility in phrasing commands. Since these keywords can also be used in variable, method, and class names, VoiceCode uses the current context to determine if the word should be used as part of a name or to create an instance of the template. Context sensitivity also allows some of these keywords to be used to accomplish differing tasks when they are uttered in different context within the source code.

VoiceCode allows local, in-screen navigation using three strategies: navigation by template, navigation by punctuation, and navigation by pseudocode. Navigation by template allows the user to move to the fields within a template, such as the conditional expression or the body of a while loop. Navigation by punctuation allows the user to move to locations before or after a nearby punctuation mark, such as "after the next parenthesis". Navigation by pseudocode involves using a keyword like "before" or "after" followed by an utterance that might be used to create the destination's source code, such as "after clients array at index 0". All of these navigation commands can be repeated multiple times in either direction, such as "again three times".

VoiceCode includes two error correcting modes: "not what I said" and "not what I meant". If the speech recognition system fails to correctly recognize the utterance, the "not what I said" mode is used to correct the misrecognition. The "not what I meant" mode is used to correct errors when the speech is recognized correctly but incorrect code is produced. For instance, "not what I meant" mode is used if VoiceCode translated the phrase "current record number" into the variable name *current_record_number* when the name *curRecNum* was desired. VoiceCode learns from these mistakes to prevent future repetitions.

VoiceCode has not undergone any user testing.

2.3 Wizard of Oz Studies

Wizard of Oz studies are commonly used to test concepts and designs for spoken language interfaces for computers. Hirschman and others [9] describe the collection, annotation, and distribution of data collected at multiple sites for a Wizard of Oz study of a spoken language interface for accessing an airline travel database. One interesting aspect of this paper is the efforts being made to verify that the data collected at the different sites can be evaluated in a consistent manner.

Many papers describe the value of conducting a Wizard of Oz study before implementing a new spoken language interface. For example, Lewin and others [12] used a Wizard of Oz study to determine the nature of sentence parsing that would be necessary for a spoken language interface to a web-based travel information system. Walker and others [25] used a Wizard of Oz study to determine the features needed for a spoken language e-mail access system.

2.4 Speech Disfluencies

Spoken language differs from written language in many ways. For instance, grammar is less formal and the context of the language is often implied. Additionally, spoken language often contains disfluencies. A disfluency is defined to be a contiguous portion of an utterance that must be removed to achieve the speaker's intended utterance. For instance, if a speaker said, "I would like to fly to New York no to Boston on Monday", then the intended utterance is "I would like to fly to Boston on Monday." The portion of the utterance, "to New York no" is a disfluency and must be removed from the original utterance to achieve the intended utterance.

The nature of disfluencies in spontaneous human speech change with the domain of the utterance as well as from speaker to speaker [14] [23]. In the domain of human to computer speech, the disfluencies can interfere with the ability of the computer to extract information from the utterance. Thus, for a spoken language interface for programming, it is important to understand the nature of the disfluencies and how they will affect the information the program receives. In this section, I will define

and discuss the types of disfluencies that were identified by previous researchers, following the terminology of Shriberg [23].

A disfluency comprises four elements:

- *Reparandum:* the entire portion of the utterance to be removed.
- *Interruption point:* the point at which the speaker interrupts the flow of the utterance.
- *Repair:* the region where fluent speech recommences, fixing the speaker's mistake.
- *Interregnum:* the portion of the utterance between the interruption point and the onset of the repair. Essentially, the interregnum is the portion of the utterance, following the interruption point, in which the words and sounds uttered have no correspondence with words uttered before the interruption point. The interregnum may include editing phrases and filled pauses.

In the example "I want to fly to New York no to Boston on Monday", the speaker repairs his utterance in order to correct the destination for his flight. The flow of the utterance is disrupted between the phrase "to New York" and the word "no"—this is the *interruption point*. The correspondence of the two prepositional phrases, "to New York" and "to Boston", on either side of the interruption point indicates that the phrase "to Boston" is the *repair* of the phrase "to New York". The *interregnum* is the collection of words and sounds uttered which follow the interruption point and precede the onset of the repair—in this case, the editing phrase "no". The *reparandum* includes the words/phrases being repaired ("to new York") and any contents of the interregnum ("no"). Thus, the reparandum for this utterance contains "to New York no". Removing the reparandum from the *original utterance* (e.g., "I want to fly to New York no to Boston on Monday") results in the *effective utterance* (E.g., "I want to fly to Boston on Monday"). Since the

purpose of removing the disfluency is to obtain the intended utterance, the effective utterance represents the intended utterance.[1]

Numerous classification schemes exist for disfluencies (cf. [23]). For the purposes of my work, I categorized disfluencies into five groups: *unfilled pauses*, *filled pauses*, *false starts*, *repetitions*, and *repairs*. Unfilled pauses are periods of time when no sounds are being uttered. While these periods of silence can cause problems for speech recognition systems, they pose no problems for a natural language processing system, and so will be discussed no further. In the following paragraphs, I will discuss the four remaining disfluency types. I took the examples from transcripts derived from the Wizard of Oz study that will be described in Chapter 4.

Filled pauses are sounds uttered that contain no semantic content, such as um, er, uh, etc. For instance, given the original utterance:

"move down to the uh student class."

the speaker pauses to determine the destination within the source code for the move, and fills the pause with the sound "uh". The interruption point is located between the phrase "move down to the" and the filled pause "uh", while the filled pause "uh" comprises the interregnum. There is no repair, since no words preceding the interruption point are modified. Removing this disfluency results in the effective utterance, "move down to the student class"

In the second type of disfluency, called a a false start, the speaker discards previously uttered content and the topic of the new utterance differs substantially from the discarded material. For example, the following user request contains a false start:

"head field equals nah go to the parentheses"

This request begins by creating an assignment statement, but the user discards the assignment and chooses to move to an existing location within the source code.

[1] I will use the terms *intended utterance* and *effective utterance* interchangeably.

The portion of the utterance describing an assignment statement must be removed to obtain the speaker's intended utterance. This phrase, "head field equals nah", is the reparandum.

The third type of disfluency, repetition, occurs when the speaker exactly repeats some portion of an utterance. Here is an example of one of these repetitions:

"print to screen print to screen quote slash quote"

In this case, the reparandum is the first instance of the phrase "print to screen", while the repair is the second instance of this same phrase. Removing the first instance of the repeated phrase results in the effective utterance "print to screen quote slash quote". Note that not every exact repetition of a phrase is a disfluency. For instance, a person could say "greater than sign greater than sign". While in some instances this could be a repetition disfluency, in other instances the speaker may be referring to the input stream operator ">>".

The final type of disfluency I will address is repairs. A repair occurs when there is a deletion from, insertion into, or modification of what was previously uttered. Generally, a repair occurs in a restatement of some portion of the utterance, although this is not always true. For instance, in the first disfluency example, "I want to fly to New York no to Boston on Monday", the preposition "to" is repeated in the repair. However, if the utterance were "I want to fly to New York no Boston on Monday", the repair would occur without repeated text.

2.5 Chapter Summary

Technology exists to allow persons with disabilities to use computers. However, these technologies are not well suited for writing and debugging computer programs. Screen readers do not provide persons with vision loss access to the same information available to sighted programmers within a programming development environment. Debugging programs becomes more difficult as a result. Persons with mobility impairments have difficulty inputting syntactically complex source code. For both of these groups, a syntax free method of programming would be beneficial.

Previous work in syntax-free programming spans three fields. Automatic programming seeks to create programs from an end-user's nontechnical specifications. This goal has not yet been reached. Structure editors guarantee correct syntax by requiring the user to create and modify source code by selecting editing options from menus and providing the required data in templates. These interfaces are cumbersome for the vision-impaired. Several small-scale natural language-based interfaces for programming have been developed.

Wizard of Oz studies have often been used to determine how potential users would interact with a new program. These studies are commonly used with spoken language interfaces to determine the nature of the speech to be processed.

Spoken language often contains contiguous portions of utterances which must be removed to achieve the speakers intended utterance. These artifacts, called disfluencies, can be divided into four types: filled pauses, false starts, repetitions, and repairs. Each of these types of disfluencies can interfere with a computer program's ability to process spoken language input.

CHAPTER 3

THE NATURALJAVA PROTOTYPE

I wanted to ensure that it was possible to create Java programs using natural language before I began this research. Therefore, I implemented a prototype which takes English sentences and sentence fragments typed on a keyboard and generates Java source code. Before I discuss the Wizard of Oz study that is the focus of this thesis, I will discuss this prototype and describe its limitations.

The prototype interface is fully implemented and can be used to produce Java source code. During a programming session, the interface comprises three text areas, an edit box, and a prompt (Figure 3.1). The largest text area displays the evolving source code. Beneath this text area is a prompt indicating that the program is processing a request or waiting for input from the user. The user types requests to the system in the edit box below the prompt. The text area at the bottom of the window shows error messages and any requests the program is making to the user (such as, "What is the name of the index variable for this loop?"). The text area along the right side of the window provides information requested by the user (such as the names and parameters for methods within a class or associated with an object) or a list of variables in scope.

Two faculty members have used NaturalJava to write exactly the same program. The first user, Joe Zachary, defined a priority queue class, and the second user, Ellen Riloff, tried to generate exactly the same source code while using different natural language sentences. Excerpts from the transcripts of the user sessions are shown in Figures 3.2 and 3.3, and the Java code that resulted is shown in Figure 3.4. NaturalJava performs capitalization correction. Thus, variable, method, and class names in the user transcripts that are not correctly capitalized appear with correct

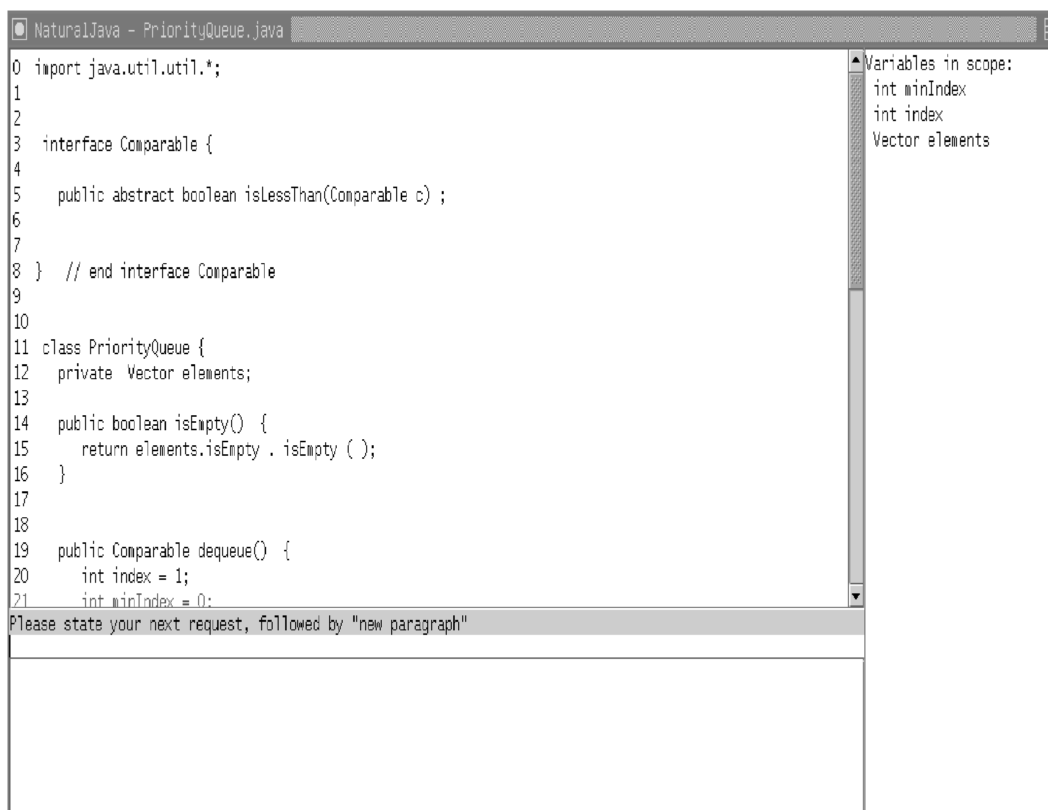

Figure 3.1. The NaturalJava user interface. The largest of the three text areas contains the evolving Java source code. The text area at the bottom of the screen holds error messages and requests for more information from the user. The text area along the right side of the window provides other data requested by the user, such as the variables in scope or the methods associated with a class.

capitalization in the source code.

In the next section, I will discuss the architecture of this prototype. In Section 3.2, I will discuss its current limitations. Section 3.3 discusses the effort that would be needed to convert NaturalJava to work with other programming languages.

3.1 The Architecture of the Prototype

The NaturalJava user interface has three components. The first component is Sundance [22], a natural language processing system that uses a shallow parser to perform information extraction [20] [19]. For NaturalJava, Sundance extracts information from English sentences and generates case frames representing pro-

1. I need to create an interface called Comparable.
2. Declare a public abstract method called isLessThan.
3. It takes a comparable called c.
4. Islessthan returns a boolean.
5. Exit the interface.
6. Make a class called PriorityQueue.
7. I need a vector and name it elements.
8. It is private.
9. Give me a constructor.
10. Elements is equal to a new vector.
11. Exit the method.
12. Make it public.
13. Declare a public method called enqueue which takes a comparable called newElement.
14. Call elements' addelement and pass it newelement.
15. Exit.
16. Create a public method called dequeue.
17. It returns a comparable.
18. Define an int called index and initialize it to 1.
19. Declare an int called minIndex equal to 0.
20. Create a comparable called minValue which is equal to elements' firstelement cast to a comparable.
21. Loop while index is less than elements' size.
22. Make a comparable called currentElement, which is equal to elements' elementat passed index, and cast to a comparable.
23. If currentelement's islessthan passed minvalue.
24. Assign index to minindex.
25. Minvalue is equal to currentelement.
26. pop out.
27. Increment index.
28. Leave the loop.
29. Call elements' removeelementat passed minindex.
30. I need to return minvalue.
31. Exit.
32. Declare a method named isEmpty which returns a boolean.
33. Isempty is public.
34. Elements' isempty is returned.

Figure 3.2. Input from User 1's NaturalJava session to create a priority queue class. Note that NaturalJava does capitalization correction on class, method, and variable names.

1. make me an interface named Comparable
2. create a method called le
3. make le public and make le abstract
4. le should return a boolean and take a Comparable parameter called c
5. exit
6. make a public class that is called PQ
7. create a private Vector and call it elements
8. give me a public constructor
9. elements gets a new Vector
10. pop out of the method
11. I want a public method that is named enq
12. This method should take a Comparable parameter called c
13. call elements' addElement and pass it c as a parameter
14. pop
15. I would like to define a public method that is named deq and that returns a Comparable
16. declare an int variable named i that is initialized to 1
17. declare an integer variable named minIndex that has an initial value of 0
18. add a Comparable variable named minValue which is equal to elements' firstElement but that is cast to a Comparable
19. declare a loop and have it iterate while i < elements' size
20. add a Comparable named c, initialize it to elements' elementAt, pass in i, and cast to a Comparable
21. if c's le when passed minvalue
22. minindex gets i
23. minvalue gets c
24. exit the loop
25. call elements' removeElementAt and pass it minIndex
26. please return minValue
27. jump out of this method
28. make a public method, name it isEmpty, and have it return a boolean
29. please have it return elements' isEmpty

Figure 3.3. Input from User 2's NaturalJava session to create a priority queue class. Note the differences between these instructions and those in Figure 3.2, which both create the same priority queue class.

```
import java.util.*;

 interface Comparable {
   abstract  public boolean isLessThan(Comparable c) ;
 }   // end interface Comparable

 class PriorityQueue {
   private Vector elements;

   public boolean isEmpty()  {
      return elements.isEmpty(  );
   }

   public Comparable dequeue()  {
      int index = 1;
      int minIndex = 0;
      Comparable minValue = (Comparable)elements.firstElement(  );
      while ( index<elements.size(  ) ) {
         Comparable currentElement = (Comparable)elements.elementAt( index );
         if ( currentElement.isLessThan( minValue ) ) {
            minIndex = index;
            minValue = currentElement;
         }
         index++;
      }
      elements.removeElementAt( minIndex );
      return minValue;
   }

   public void enqueue(Comparable newElement)  {
      elements.addElement( newElement );
   }

   public  PriorityQueue()  {
      elements = new Vector(  );
   }
}   // end class PriorityQueue
```

Figure 3.4. Output from a NaturalJava session. The input sentences shown in Figures 3.2 and 3.3 result in the Java source code seen above.

gramming concepts. The second component is PRISM, a knowledge-based case frame interpreter that uses a manually constructed decision tree to infer high-level editing operations from the case frames. The third component is TreeFace, an abstract syntax tree (AST) manager [17]. PRISM uses TreeFace to manage the syntax tree of the program being constructed.

Figure 3.5 illustrates the dependencies among the three modules and the user. PRISM presents a graphical interface to the user, who types an English sentence describing a program construction command or editing directive. PRISM passes the sentence to Sundance, which returns a set of case frames that extract the key concepts of the sentence. PRISM analyzes the case frames and determines the appropriate program construction and editing operations, which it carries out by making calls to TreeFace. TreeFace maintains an internal AST representation of the evolving program. After each operation, TreeFace transforms the syntax tree into Java source code and makes it available to PRISM. PRISM displays this source code to the user, and saves it to a file when the session terminates.

In the next three sections, I will discuss each of these modules in more detail.

3.1.1 Sundance

I selected English as the input language because it would not require the student to learn a new language to begin programming. He or she can use the language the instructor is using in class to write programs. However, English has two drawbacks as an input language: natural language specifications can be ambiguous and incomplete, and natural language processing can be fragile because complete natural language understanding is still beyond the state of the art. I addressed the first problem by limiting the role of inference in my system. I decided that user requests, while stated in English, must be very similar to programming constructs. This level of specification is relatively well defined, yet general enough that the programmer can focus on programming rather than syntax. The interface can detect when a command is incomplete (e.g., the terminating condition of a loop is missing) and prompt the user, but the role of inference in NaturalJava is mainly limited to the disambiguation of general verbs (e.g., “add” can refer to arithmetic

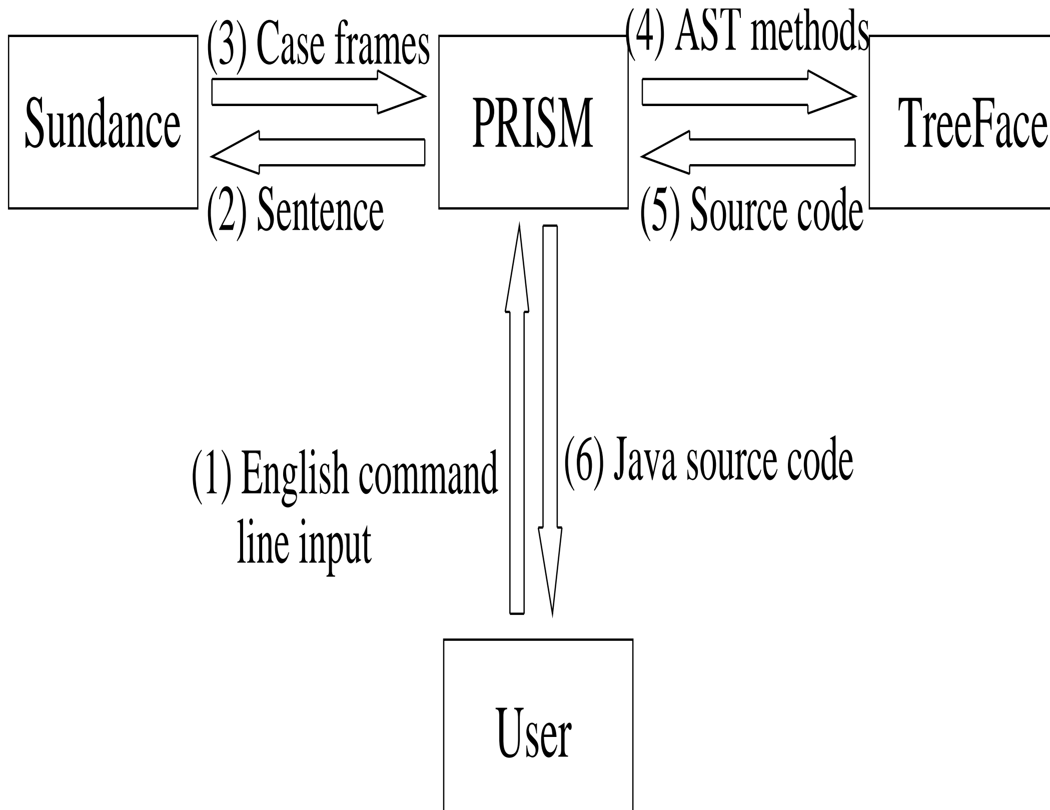

Figure 3.5. Architecture of the NaturalJava prototype.

or insertion).

I addressed the second problem of fragile natural language processing by using information extraction technology supported by a partial parser. Partial parsers are typically more robust and flexible than full parsers, which try to generate a complete parse tree for each sentence. Full parsers often fail on sentences that are ill-constructed or ungrammatical. Partial parsers are more robust because they do not have to generate a complete parse structure, but instead generate a syntactic representation of sentence fragments.

Information extraction (IE) is a form of natural language processing that involves extracting predefined types of information from natural language text [3] [21]. The goal is to identify information that is relevant to the task at hand while ignoring irrelevant information. Information extraction systems have been

built for a variety of domains, including Latin American terrorism [20] [19], joint ventures [19], microelectronics [19], job postings [2], rental ads [24], and seminar announcements [7].

For the NaturalJava interface, I used IE techniques to extract information related to Java programming constructs from the user's input. The natural language engine used by NaturalJava is a partial parser called Sundance [22]. Sundance generates a flat syntactic representation of sentences and also can activate and instantiate pattern-based templates, or case frames. Initially, I manually designed 400 case frames to extract information about relevant programming constructs. Using the data collected during the Wizard of Oz study (Chapter 4), another graduate student and I expanded this set to 500 case frames. Sundance generates 39 types of case frames for NaturalJava; the nature of these types are summarized in Table 3.1.

As an example, consider the sentence "Create a for loop that iterates from 1 to 10." Sundance begins by deriving a partial parse for this sentence, which includes part-of-speech disambiguation, syntactic bracketing, clause segmentation, and syntactic role assignment. Sundance then instantiates all active case frames to extract information from the sentence. The case frames represent local linguistic expressions primarily revolving around verbs and nouns. Each case frame has a trigger word and an activating function that determines when it is applicable. For example, a case frame might be triggered by the word "iterates" when it appears as an active voice verb form. A case frame also has a type, which represents its general concept, and an arbitrary number of slots that extract information from local syntactic constituents.

Figure 3.6 shows a case frame triggered by the verb "iterates." It contains four slots that extract information from the subject of the clause and from three prepositional phrases. For example, the subject of the clause will be extracted as a CONTROL_FLOW object, while the object of the preposition "from" will be extracted as the start condition for the loop. The prepositional phrases may appear in any order, and any subset of these slots may be instantiated, depending on the

Table 3.1. Summary of the case frame types used by Sundance in NaturalJava.

Case frame type	Purpose	Example trigger words
allocate	Allocating memory	new
array_length	Declaring array size	size, length
array_offset	Indicates the offset in an array	sub, index
assignment_comparison	Indicates an assignment	equals
	or value comparison	set
cast	Casting an object	cast
comment	Indicates presence of a comment	comment, precondition
comparison	Comparing two values	less than, greater than
conditional	Conditional expression	if, then
conjunction	Joins elements of conditional exps.	and if, or if
construct	Indicates programming construct	class, method
control_exp	Expressions controlling	controlled
	entry to blocks	expression
control_flow	Control flow structures	loop, iterate
create	Declaring variables, methods, etc.	create, declare
debug	Debugging features for NaturalJava	trace, debug
edit	Modify source code	delete, change
extends	Inheritance	extends, inherits
i_o	Input / Output	open, save, print
implements	Implmenting interfaces	implements, uses
info	Requesting information	list, show
insertion_mode	Changing where new	insert
	code is placed	append
list	Requesting information	"a list of . . . "
literal_string	Declaring a literal string	says
location	Indicate locations in code	top, end
loop	Nouns indicating a loop	for loop, while loop
math	Math operations	multiply, plus
method_calls	Invoking a method	apply, call
multi_purpose	Usable in more than one context	add, make
name	Naming a variable, method, etc.	called, name
navigation	Move to a new location	go, move
negation	Negating a value	minus, negative
packages	Creating and importing packages	import, package
param_passing	Parameters for methods	pass, takes
pop_out	Navigating up AST	pop, exit
property	Properties of classes, members	is private, are static
property_name	Properties of classes and members	public, final
push_into	Navigating down AST	push, enter
type_list	Types of things listed as info	methods, parameters
type_name	Names of intrinsic types	boolean, double
value	Indicates a value following	value

```
CF
Name: iterates
Act_Fcns: active_verb(ITERATES)
Type: control_flow
    Slot: subj <- object
    Slot: PP(FROM) <- loop_start
    Slot: PP(TO) <- loop_end
    Slot: PP(WHILE) <- exit_condition
```

Figure 3.6. An example of a case frame in Sundance. This case frame is instantiated when the word "iterates" is used as an active voice verb construction.

input sentence.

The final output of Sundance for the example sentence is shown in Figure 3.7. Two case frames are generated, representing a CREATE concept and a CONTROL_FLOW concept. The CREATE case frame indicates that a for loop should be created, and the CONTROL_FLOW case frame specifies the control conditions for the loop. Figure 3.7 shows the instantiated version of the case frame shown in Figure 3.6. Notice that Sundance did not extract an exit condition because there was no prepositional phrase for the preposition "while" in the sentence.

3.1.2 PRISM

The Programming Instruction Synthesis Module (PRISM) forms the core of NaturalJava. It provides the user interface (Figure 3.1) and ties all of the components together. PRISM receives the case frames produced by Sundance, determines the nature of the action requested by the user, collects the necessary information to carry out that action, and calls the appropriate methods within TreeFace in order to modify the evolving source code.

PRISM divides the case frame processing into two tasks. First, it determines the type of action the user desires. Then, it retrieves the necessary information from the case frames to carry out that request. PRISM makes one assumption to simplify the task of determining the action to be taken: PRISM assumes that

```
Trigger: create
Type: create
    create_type: "a for loop"

Trigger: iterates
Type: control_flow
    object: "a for loop"
    loop_start: "1"
    loop_end: "10"
```

Figure 3.7. The instantiated case frames produced by Sundance given the sentence "Create a for loop which iterates from 1 to 10."

each request by the user represents only one type of action taken on the AST. For example, a single request can contain either a request to move to a new location in the source code ("Move down two lines") or the declaration of a variable ("Declare an int called index"), but it cannot request the program to move to a new location and declare a variable at that location ("Move down two lines and declare an int called index") in a single command.

PRISM's first step is to determine the nature of the user's request. PRISM determines this in two stages. First, PRISM utilizes a heuristic function that examines the types of all of the case frames returned by Sundance to classify user requests into the possible types of AST actions, such as declaration, navigation, or I/O. Some case frames are triggered by verbs that can be used in more than one type of command, such as "make" and "give". These "multipurpose" case frames are examined with a concept disambiguation method. This method examines information in the extracted strings to determine the type of concept these case frames represent and relabels their types accordingly. For example, PRISM determines that "make a double called myDouble" is a variable declaration, and the case frame triggered by "make" is relabeled with type *create*. However, "make myName public" changes a property of a data member, so PRISM relabels the

case frame triggered by “make” with type *property*. If the multipurpose case frame does not contain the necessary information, PRISM discards it. For example, given “make x equal to y,” PRISM discards the “make” case frame because its extracted string, the direct object phrase “x”, does not contain any content indicating the nature of an action. Then, PRISM examines the remaining case frame instantiated for “equal” and can classify the request as an assignment.

I developed this heuristic function to allow the user to phrase requests in a flexible way. NLC [1] and Moon [26] both required that requests be structured so that the first verb defined the nature of the action to be taken. With this restriction, requesting an assignment statement with “Assign x plus y to z” would be a legal request, but “add x to y and assign it to z” would not be legal. Since both requests represent a single AST action (i.e., creation of a single assignment statement), my heuristic function allows both phrasings of the request to be legal.

PRISM uses the classification provided by the heuristic function as the first decision made in a manually constructed decision tree that converts the case frames extracted by Sundance into actions to be taken on the AST. Subsequent levels of the decision tree examine the case frames’ trigger words and extracted strings to further subdivide the command. For example, the heuristic function determines the request to be a declaration, and subsequent levels of the decision tree determine if the request is declaring a class, method, data member, or local variable. Additionally, PRISM often uses the current editing context of the AST to further constrain the nature of the user’s request.

Once the decision tree fully classifies the requested action, PRISM gathers the necessary information from the extracted strings held in the case frames. Given this information and the nature of the action to be taken, PRISM calls the appropriate methods within TreeFace to complete the user request.

3.1.3 TreeFace

The third component of NaturalJava is TreeFace, which creates and manipulates objects that encapsulate AST representations of Java source files.[1] TreeFace provides constructors that create empty ASTs and that initialize ASTs by parsing Java source files. TreeFace also provides methods that navigate through, add content to, perform generic editing operations on, and return information about an AST. In response to instantiated case frames produced by Sundance, PRISM composes appropriate sequences of TreeFace constructor and method invocations.

A TreeFace object also keeps track of the current editing context. PRISM uses this context to determine where in an AST a particular editing operation should take effect. The user must often change the editing context, much as the user of a standard editor must often change the current selection. Since the editing context is always a subtree of the entire AST, changes to the editing context can be expressed in terms of motion through a tree. However, TreeFace also maintains information about the line numbers for each node in the AST. Therefore, navigation can also occur with respect to the lines shown in the code window.

TreeFace provides content creation methods that create new classes and interfaces, member variables, methods, local variables, compound statements such as loops and conditionals, and simple statements such as assignments and returns. It also provides methods that allow the user to change certain attributes of existing constructs. For example, the user can make a member private.

TreeFace's generic editing operations allow the user to delete the current selection and to undo recent modifications to the AST. TreeFace also provides operations that report the state of the AST. These operations allow the user to request information about the AST, such as the list of variables currently in scope. PRISM uses this capability to answer questions posed by the user.

---

[1]The underlying AST was generated using Java Compiler Compiler (JavaCC).

3.2 The Limitations of the Prototype

The NaturalJava prototype possesses a number of limitations. Its interface is very simple. All input must be entered in English sentences—the mouse cannot be used for navigation or selection of program constructs. As was described previously, each user request is limited to containing only one type of AST action. Additionally, compilation and debugging facilities are not integrated into the interface.

NaturalJava's language is limited in several ways. Its vocabulary is limited by incomplete case frame coverage of all of the concepts utilized in creating and editing source code. Sundance has problems correctly parsing some of the unusual sentence structures that might be used in programming. For instance, verbs used as method names cannot be extracted. Given the sentence "Declare a method called add", Sundance is unable to instantiate the case frame triggered by "called" because "add" is misparsed as a verb but should have been parsed as a noun in this context.

The prototype is best suited for writing new source code and doing local, statement-level editing. Expression-level editing and global program modifications are unsupported. For example, the only way to modify an expression is to delete and replace the statement that contains it. Renaming a variable requires replacing its declaration as well as every occurrence of it.

NaturalJava supports a large but incomplete subset of Java. It does not support nested classes and packages because I have not yet built the required AST support into TreeFace or PRISM. Also, NaturalJava currently supports only four classes from the Java API (e.g., Object, String, Integer, and Vector). Such limitations are a result of my depth-first development strategy.

3.3 Converting to Other Programming Languages

NaturalJava currently produces source code in the Java programming language. Converting NaturalJava to produce source code in other programming languages would require substantial effort. Aspects of each of the three principal components of NaturalJava would need to be modified in order to carry out such a conversion.

Languages that include programming constructs and concepts different from those found in Java would require modifications to both Sundance and PRISM. New vocabulary would need to be added to Sundance's dictionaries and new case frames generated to encapsulate these new concepts. In PRISM, new keywords would need to be added to the decision tree and new branches created to identify and process the new concepts. Additionally, new methods in PRISM would be required to correctly extract the necessary data for these new constructs.

For instance, languages such as C and $C++$ utilize pointers to locations in memory. Java does not have the ability to access memory in this fashion. As a result, all of the language used to describe pointers would need to be integrated into Sundance and PRISM. New verbs such as "point" and "dereference" would have to be added to the Sundance dictionaries. Sundance case frames triggered by verbs such as "points" and "dereference" would then need to be generated to capture the concepts essential to using pointers. In PRISM, new methods would need to be developed to process these new case frames and properly extract the information they contain. Additionally, new keywords, such as "pointer" and "address", and the branches these words might indicate would need to be added to PRISM's decision tree in order to properly determine the correct actions to be taken during processing.

Any new language would require major modifications to TreeFace. The underlying AST implementation would have to be replaced with an AST implementation conforming to the new language's grammar. All of the methods which create, modify, and query the AST would have to be modified to correctly access the components of the new AST implementation. new methods would be required to manipulate constructs found in the new language that were not found in Java. Essentially, the interface for TreeFace would remain, but the underlying implementation would require extensive modification.

3.4 Chapter Summary

The NaturalJava prototype accepts English sentences and sentence fragments from the keyboard and produces syntactically-correct Java source code. The proto-

type comprises three components: Sundance, PRISM, and TreeFace. Sundance performs robust information extraction using a shallow parser. Information is extracted based upon the occurrence of predefined keywords, primarily verbs and nouns. Information relevant to these keyword triggers is extracted from surrounding phrases and returned in data structures, called case frames. PRISM, a knowledge-based case frame interpreter, examines the data held within these case frames and determines the type of action contained within the request. Once the nature of the request is determined, PRISM collects the needed information from the case frames to carry out that request. PRISM calls methods within TreeFace, an abstract syntax tree manager, to place the necessary information into an abstract syntax tree representing the evolving source code. TreeFace manages the creation and modification of these abstract syntax trees, and provides information to PRISM on the current state of these trees. Many of NaturalJava's limitations result from the depth-first approach used during development. Other limitations result from a lack of vocabulary coverage and knowledge of how users would interact with such a system.

Converting NaturalJava to produce source code in a different programming language would require substantial effort. Sundance and PRISM would need to be modified in order to correctly process any new programming constructs present in the new target language. TreeFace would need to be heavily modified to work properly with the abstract syntax trees of the new target language.

CHAPTER 4

THE DESIGN OF A WIZARD OF OZ STUDY

The NaturalJava prototype demonstrated that it is possible to build a Java programming system with a written natural language interface, albeit one with the limitations described in Section 3.2. My long-term goal was to build a fully-featured Java programming system with a spoken language interface. But, before I could contemplate designing such a system, I realized that I needed to better understand how potential users would use such a system.

In particular, I wanted to investigate three specific issues relevant to the development of a spoken language interface for programming. These issues are the types of commands requested of the interface, the vocabulary used in these requests, and the effect that disfluent speech would have on such an interface. Even if I had somehow managed to graft a perfect speech recognition system to the prototype's front end, further experiments with the prototype would not have been helpful in answering these questions. The limitations of the prototype would have made it impossible to study these issues. What I needed was a way to observe programmers using a fully-featured spoken language interface for programming without having to first build the system. My solution was to design and conduct a *Wizard of Oz* study.

In a Wizard of Oz study, a test subject is asked to experiment with a putative software system running on a computer. Although the system presents what appears to be a fully-functional user interface, that interface exists only to send user input to a second computer. An expert, called the *wizard*, sits at the second computer in another room. The wizard examines the user input from the subject's

computer and determines what the subject is trying to do. The wizard then transmits the appropriate response back to the system on the subject's computer, which displays the results to the subject. In this way, the subject is presented with the illusion of a fully-functional system.

I designed and conducted the study over a period of approximately 15 months. Between October 2000 and May 2001, two undergraduate research assistants and I implemented and tested the hardware and software infrastructure required for the study. During the summer semester of 2001, a single subject from an introductory Java programming class tested this infrastructure. During the fall semester of 2001, eight subjects from an introductory C++ programming class participated in a Wizard of Oz study. (I chose students from a C++ class because no Java class was offered; fortunately for my goals, Java and C++ are syntactically and conceptually similar.) Each subject was paid $10/hour for their time during the study.

Over the course of the semester, each subject used the system once a week for two hours. During each session, the subject would typically use the system to work on a homework assignment from his or her programming class. If the subject had begun work on the assignment before the session, he or she would e-mail the source code to us as a starting point. At the completion of each session, we e-mailed the resulting source code back to the subject.

4.1 Subjects

I recruited subjects for this study from CPSC 2000, an introductory programming class for undergraduate students in the Department of Electrical Engineering. As described above, this class was taught using the C++ programming language.[1] I invited all students enrolled in CPSC 2000 to participate in the Wizard of Oz study. Eight students volunteered to take part, and all eight were accepted into the study.

[1] I selected this class because the introductory programming class taught the same semester for undergraduate students in the Department of Computer Science utilized the Scheme programming language—a programming language whose syntax and structure differs markedly from Java.

Recruiting subjects in this manner results in one possible bias. All of the subjects are students in the same class. As a result, the instructor could inadvertently bias the subjects in the language they use to describe programming constructs. If the instructor describes programming constructs in a consistent manner, the students may unconsciously adopt the language of the instructor for these constructs. Additionally, since they were working on the same set of assignments, the nature of these assignments, and any source code they were given, could also influence their performance in the study.

During the study, each subject sat at a computer that displayed a simple one-window user interface. The interface was divided into a code region, a prompt region, and a message region. The evolving source code appeared in the code region, which was the largest of the three. (The code region also contained a highlighted region, which indicated the point at which editing operations would apply.) The prompt region indicated whether the system was processing an input or was ready to receive the next input. The message region displayed, as necessary, messages that requested more information from the user or warned that the previous command was not understood.

The keyboard and the mouse were removed from the computer, leaving an audio headset with a boom microphone as the only input device available to the subject. The subject communicated with the system by speaking into the microphone, and the system communicated with the user by displaying source code, messages, and status updates in one of the three regions.

Before each subject's first session, I explained the purpose of the three window regions and instructed the subject to mark the end of each request by saying "new paragraph." I avoided giving more extensive instructions or examples because I did not want to bias how the subjects would use the system. Unfortunately, this strategy resulted in some subjects dictating syntax, symbol by symbol. Since we were trying to develop a syntax-free method of programming, these sessions did not provide me with interesting data.

I subsequently decided to give the subjects a way to think about the system

and one concrete example of how to use it. I told them:

> *Think of the computer as if it were one of your classmates. It knows exactly what you know, and you should give it instructions on what to write in the easiest words possible.*

I also gave them the following written example:

> *To print "Hello, world" to the screen, you can say "Print quote hello comma world quote to the screen new paragraph."*

Given the mental picture and the example, the subjects were able to work with the system as I had intended. A typical interaction with the system went as follows:

- The prompt initially reads "Please state your next request."
- The subject says "Give me a for loop that goes from zero to ten new paragraph."
- The prompt becomes "Processing, please wait" as the system determines a response.
- The message region displays the query "Name of index variable?"
- The subject says "i new paragraph".
- The following code is inserted at the cursor in the code region:

```
for (int i = 0; i <= 10; i++) { | }
```

- The initial prompt is redisplayed.

4.2 Wizards

During the study, I used three undergraduate research assistants as the wizards. During each session, one of the wizards donned headphones and sat at a computer in a different room. The subject's utterances were:

- sent to the wizard's headphones so he could hear what the student said;
- piped through a speech recognizer, whose output was both displayed to the wizard (in case he forgot what the subject said) and logged to a text file;
- recorded to an audio file.

The wizard manipulated three windows on his computer: a text editor in which he composed the source code to be displayed in the subject's code region, a text editor in which he composed messages to be displayed in the subject's message region, and a read-only window that displayed the output of the speech recognizer.

Each time the subject issued a command, the wizard responded by either modifying the source code or composing a message to be displayed in the interface's message area, as appropriate. Only when the wizard saved the modified source code, or the message to be displayed, to a file was the subject's display updated. From the point of view of the subject, the system would appear to compute for a few seconds, at which point the entire response would appear instantaneously.

All source code modifications and messages were logged in coordination with the logs of the subject's audio requests. This made it possible, during the subsequent analysis phase, to recreate the exact sequence of events of each session. See [5] for further details on the software and hardware infrastructure.

4.3 Subject Feedback

During the study, the subjects provided feedback in two forms: responses to a questionnaire and spontaneous comments during the sessions. Both of these methods of feedback proved useful.

I received a great deal of information about the features the students desired in the interface through the questionnaires. At the end of each two-hour session, I asked the subject to move to a computer with a keyboard and fill in a web-based evaluation form. The evaluation form requested feedback on whether the system was easy to use, which features worked best and worst, and what would make the

system easier to use. I will discuss the conclusions drawn from these questionnaires in Section 5.5.

The wizards and I received spontaneous verbal feedback from subjects during their sessions. The subjects often forgot they were being recorded; their unguarded comments give me insight into their impressions and expectations. Some remarks (e.g., "Wow! That's great!") exhibit surprise that the program functioned correctly. Others (e.g., "No! That's not what I wanted.") reveal when the subject perceived problems. Such comments, taken in conjunction with the related changes to the source code, have helped me sort out what users meant when they employed ambiguous language. For example, one such source of ambiguity is the word "add," which can be used to describe a mathematical operation (e.g., "add x to y") or to ask that an object be included in a larger context (e.g., "add a method to this class").

4.4 Chapter Summary

I used a Wizard of Oz study to investigate several specific issues relevant to the development of a spoken language interface for programming. In this study, subjects sat in front of an (imaginary) interface for a spoken language programming system. The only input device available to the subjects was a microphone. A few seconds after the subject uttered a request to add or modify the displayed source code, the interface presented the updated source code to the subject. In this way, the subjects believed they were testing a real spoken language interface for programming. However, the subjects never realized that the responses were actually generated by an expert programmer sitting in another room. These experts, called wizards, listened to each request from a subject, modified the source code, and placed the result back on the subject's computer via the network.

Eight introductory programming students took part in this study. These subjects interacted with the interface once a week for two hours. We collected a wide variety of data from each of these sessions, including recordings of each uttered

request, the resulting changes to the source code, and any messages provided to the subjects through the interface. The next chapter contains analysis of these data.

CHAPTER 5

WIZARD OF OZ STUDY RESULTS

Many difficult problems will need to be solved before a flexible, fully-functional spoken language interface for programming can be developed. The goal of this research is to provide an understanding of a few of the basic issues faced in this development process. In this research, I focused on a single potential group of users — novice programmers taking an introductory programming class. I investigated issues related to a spoken language interface for programming. These issues are summarized in the following research questions:

- What types of commands do the people use and how do they use them?
- Do the people share a common vocabulary or do the words they use differ substantially from person to person?
- What types of disfluencies are used by the people, how frequently do these disfluencies occur, and how do they impact the underlying natural language processing (NLP) system that extracts information from the user requests?

These three questions address three principal issues relevant to building a system to convert spoken English into source code. I will discuss the issues raised by each of these questions along with the associated results in depth later in this chapter. First, however, I will describe the data I collected during the Wizard of Oz study and the preparation of these data for analysis. Then, in Section 5.2, I discuss the types of commands the subjects used. In Section 5.3, I analyze the vocabulary used by the subjects. Section 5.4 describes the types and frequencies of disfluencies found in the programming request domain and the effect these disfluencies have on the

underlying NLP system. Finally, I draw conclusions, based on informal feedback and my own observations, about the features needed for a spoken language interface for programming in Section 5.5.

5.1 Data Collection

The Wizard of Oz study comprised 67 sessions totaling approximately 125 hours of data. During these sessions, the subjects made 8117 requests to the interface. Each session generated four types of data: an audio recording of each request, the output from the NaturallySpeaking speech recognition system for each request, any messages sent to the message region, and the sequence of changes to the source code. Following the study, two undergraduate research assistants manually transcribed the audio recording of each request to make further analysis possible.

The data for each session are organized into a single HTML file. Each entry within this file represents a single subject request and has a link to the recorded audio, the manual transcription of the request, the output from NaturallySpeaking, the changes made to the source code, and any messages from the wizard related to this request. This organization provides a convenient way to relate each request to what the wizard interpreted the request to mean. Table 5.1 shows an entry from one of these files.

A graduate student read through the transcripts and, using the organized data, manually labeled each request with one or more tags that indicate the types of commands that the request expresses. Table 5.2 gives the 24 tags used for this purpose. I selected this group of tags to divide the user requests into the different types of actions a natural language interface must address. For example, the request "Move up three lines and insert a declaration of an integer called n" would be tagged as both a navigation command and a declaration command. The set of tags that characterize a request is unordered (it does not reflect the order in which command types occur in the request) and contains no duplicates (it does not show the number of times that a particular command type occurs in the request). I chose this method for tagging the command types because order and duplication are not relevant to

Table 5.1. Example entry in collated session data file. The manual transcripts include information about pauses in the subject's speech. In this case, *[mp]* represents a medium-length pause (3–10 seconds).

Audio:	[MP3]
Manual transcript:	make a new function of type void called input names receiving the parameters int num players and the array players new paragraph [mp] array type string new paragraph
NaturallySpeaking output:	making a function of that avoid culled input name is receiving the parameters in nome players and they array players array tight string
Messages:	Array type?

Code:	void inputNames(int numplayers, string players[])
	{
	
	}

the analysis I performed.

Throughout this chapter, *word instances* refers to a collection of uttered words, while *unique words* is a synonym for the set of unique words found within that collection (i.e., all duplicate instances have been removed). In other words, a word uttered multiple times occurs multiple times in the collection of word instances, but that word occurs only once in the set of unique words. For instance, all 10 words found in the utterance "Declare a String called foo and a String called bar" comprise the word instances, but only seven words {declare, a, string, called, foo, and, bar} comprise the unique words.

Recall that I asked the subjects to end each request with "new paragraph." (These words indicated to both the wizard and the NaturallySpeaking software that the request was complete.) This phrase at the end of each request was removed before I began analyzing the data. Thus, instances of these phrases do not appear in the data set or in the results that follow.

Table 5.3 shows some high-level statistics on the data collected during the Wizard of Oz study. I developed a list of 270 stop words [10], and I removed all instances of these stop words from the data set. Excluding stop words, there

Table 5.2. The command type tags applied to the Wizard of Oz data.

Array	Declaring or accessing an array
Assign	Assigning a value to a variable
Calls	Calling a function
Cancel	User discarding previously uttered material
Cast	Type casting
Comment	Comment included in source code
Cond	Conditional expressions
Decl	Declaring a class, function, or variable
Edit	Making changes to existing source code
Extend	Deriving a class from a super class
Flow	Control flow statements, excluding loops
Implement	Implementing an interface
Info	Requesting information from the interface
Invalid	Invalid requests to the interface
I/O	Input/output within the source code
Loop	Declarations of loop constructs
Math	Math operations
Nav	Navigation within source code
Overload	Overloading methods and operators
Packages	Importing external libraries
Param	Parameter passing
Property	Declaring and modifying class, method, and data member properties
Return	Types and values returned by functions
System I/O	Input/output used by the interface

are 2,220 unique words and 65,122 word instances in the data set. About 40% of the 2,220 unique words were uttered only once.

Results from my analysis of the data set make up the remainder of this chapter. Recall that I set few limits on how the subjects could form their requests. Essentially, I asked the subjects to pose the kinds of requests that they would use when talking to a fellow student. This resulted in very noisy data with many disfluencies and, in some cases, a lot of thinking aloud.

5.2 Command Type Results

Each request uttered by a subject when creating or editing source code would require an interface to take one or more distinct actions. Some examples of these

Table 5.3. Statistics on the data set collected during the Wizard of Oz study.

	Original data set	Excluding stop words
Number of word instances	98,037	65,122
Number of unique words	2,394	2,220
Number of unique words used only once	894	880

actions include moving within the source code, declaring variables, and creating control flow constructs. I refer to each distinct type of action as a command type. For instance, here are some examples of users requesting movement within the source code (i.e., the navigation command type):

1. *"move down six lines."*
2. *"go to body of if statement."*
3. *"okay um go to the void function list students of a state by name."*

Similarly, some examples of subjects requesting declarations of classes and functions (i.e., the declaration command type) include:

1. *"create another function called list all students."*
2. *"okay the next function needs to be a void function called set ski boot of class type ski."*
3. *"class address capital a public."*

Many of these requests do not have a simple sentence structure, and many words are used in a fashion not normally seen in English text. In fact, many requests do not include a verb at all, such as the third declaration example.[1] The first step in extracting information from these requests is determining the command types

[1] The third declaration example requests the declaration of a class named "address", and the name "address" begins with a capital 'A'.

they represent. Thus, a good understanding of how command types are employed in practice is a prerequisite for designing an automated procedure for isolating and identifying the types of commands in these user requests.

A knowledge of the relative frequencies of the different command types is important because a spoken language interface for programming must accurately identify the most frequently occurring command types. Therefore, I examined the relative frequencies of the different command types. The usage frequencies of the 24 command types are shown in Table 5.4. These data show frequent uses of navigation and editing commands. In retrospect, the predominance of these command types is not surprising. As novice programmers write source code, they often forget to include necessary declarations and other statements, and so they move around in the file to add these missing pieces. Additionally, these students were often given code "templates" to modify for their assignments. Making these changes often required a great deal of moving through and editing existing lines of source code. The pattern of usage shown in these numbers for all subjects (Table 5.4) parallels the patterns of command usage displayed for each individual subject. While the ordering of some of the less frequently used commands changes between users, the pattern remains the same.

Table 5.4. Distribution of command type usage.

Command type	Usage
Navigation	24.1%
Edit	21.8%
Declaration	7.7%
I/O	6.9%
Parameter passing	6.4%
Comment	5.3%
Method calls	4.8%
System I/O	4.0%
Assignments	3.8%
Other	15.2%

The presence of more than one command type within a single utterance further complicates the classification of user requests. For instance, the request “move up 3 lines” contains only one command type—a navigation command. However, “go to the first line in this method and change i to index” contains two command types—a navigation command and an edit command. If multiple command types routinely co-occur within requests, it would greatly increase the complexity of the process needed to identify the command types present within a request. Table 5.5 classifies each request based on the number of different command types found within it. These data show that approximately 90% of the requests contain only one or two types of commands, and 97% of the requests contain three or fewer types of commands. This suggests that an automated procedure for classifying command types need not be overly concerned with complicated requests, and that it would not be very constraining to limit users to no more than two or three command types per request. However, it is worth noting that the presence of multiple instances of a command type within a single request may add some complexity to the classification task.

Another way to view the complexity of the user requests is to examine the different command types contained within each request. I use the term “command tag cluster” to denote the set of all command tags associated with a single request. For instance, the command “move to the first line in the method and change i to index” contains both a navigation command and an edit command. Therefore, “nav:edit” represents the command tag cluster for this request. If many different

Table 5.5. Frequency of the number of different command types per user request.

Number	Usage
1	61.0%
2	28.5%
3	7.7%
4	2.2%
5–7	0.6%

command tag clusters commonly occur in user requests, identifying the command types present within the request would be more difficult. Alternatively, a small number of commonly occurring command tag clusters may simplify recognizing them.

To examine the number of command tag clusters used by the subjects, I sorted the clusters based on the number of times each cluster occurred within the data. The subjects used 349 different command tag clusters. Table 5.6 shows the 15 most common command tag clusters and the percentage of usage they comprise. These 15 tag clusters make up 75% of all user requests. In each of these clusters, only one or two command types appear. This preponderance of simple commands would make classifying the types of commands within the majority of requests much easier.

Table 5.6. The 15 most common command tag clusters, without regard to tag order. The number of times each cluster occurs, and this number as a percentage of all commands in the corpus, are given. These 15 command tag clusters account for 75% of all command tag clusters.

Command Tags	Number	Percentage
nav	1421	17.7
edit	1302	16.3
edit:nav	562	7.0
i_o	556	6.9
decl	452	5.6
sys_i_o	301	3.8
comment	274	3.4
comment:edit	219	2.7
packages	189	2.4
calls:param	147	1.8
comment:nav	147	1.8
decl:param	145	1.8
return	115	1.4
cond	98	1.2
nav:sys_i_o	92	1.1

While these few command tag clusters comprise 75% of the requests, identifying the remaining command tag clusters could be difficult. Determining that some command types occur most commonly alone may provide a useful indicator when classifying the command types found within a request. Similarly, knowledge that certain command types most commonly occur with other command types could also aid in this classification. I examined each request containing a given command tag and tabulated the number of other command types present within the request.

Table 5.7 shows how often additional command tags occurred in combination with each of the 23 other command types. For example, a navigation command (nav) occurs 49.2% of the time as the only command type within a request. However, 36.5% of the navigation commands occur with one other command type within a single request, while 10.6% of the requests containing navigation commands also contain two other command types. (Note that these numbers are percentages of a single command type's total number of occurrences.) These data show that all command types commonly co-occur with other command types. Four of the command types (i_o, overload, property, and sys_i_o) occur alone within a request more than half of the time. In contrast, four command types (array, extend, implement, and math) are virtually never used by themselves.

In summary, these data show that the subjects tended to make simple requests of the interface. Their requests predominantly contained only one or two command types. Additionally, the subjects used a small subset of command types for the bulk of their requests, and 75% of the requests comprise a small number of command tag clusters. These results suggest that determining the types of actions being requested by novice programmers is not an overly complex task. One aspect of the study may bias these results. The small number of subjects, working on a small number of tasks, may not provide a large enough sample size to provide reliable results.

Table 5.7. Number of tags present within a request as a percentage of a given command tag's usage.

#tags in request	1	2	3	4	5	6	7
array	0.0	25.5	44.4	17.7	10.0	2.2	0.0
assign	15.6	35.8	23.4	17.9	4.9	2.0	0.1
calls	8.5	41.1	34.6	12.1	2.5	0.9	0.0
cancel	4.9	36.6	34.1	16.3	3.9	3.9	0.0
cast	25.0	0.0	25.0	50.0	0.0	0.0	0.0
comment	37.2	51.6	9.6	0.9	0.4	0.1	0.0
cond	27.1	45.7	19.9	4.9	1.6	0.2	0.2
decl	45.3	32.0	13.0	6.4	1.9	0.9	0.0
edit	47.5	40.5	8.8	2.2	0.5	0.1	0.0
extend	0.0	33.3	33.3	33.3	0.0	0.0	0.0
flow	44.4	33.3	16.6	0.0	0.0	0.0	5.5
i_o	58.4	22.9	14.1	2.9	0.8	0.5	0.1
implement	0.0	66.6	33.3	0.0	0.0	0.0	0.0
info	22.2	33.3	44.4	0.0	0.0	0.0	0.0
loop	10.0	22.6	21.3	31.4	10.6	3.1	0.6
math	0.8	44.9	28.7	18.2	5.5	1.4	0.2
nav	49.2	36.5	10.6	2.8	0.4	0.1	0.0
overload	10.0	50.0	40.0	0.0	0.0	0.0	0.0
packages	80.1	16.4	2.9	0.4	0.0	0.0	0.0
param	10.4	48.1	29.9	9.0	1.6	0.6	0.0
property	51.4	24.6	20.1	2.9	0.0	0.7	0.0
return	42.1	33.3	13.5	8.4	1.4	1.0	0.0
sys_i_o	59.1	30.5	7.3	2.5	0.1	0.1	0.0

5.3 Vocabulary Results

The vocabulary used by the subjects must be understandable by any natural language interface for programming. Consequently, gaining insight about the range of vocabulary across subjects would greatly impact how any future interfaces are designed. If all of the subjects use a consistent vocabulary, future systems can be designed around this "core vocabulary". If the vocabulary varies widely between subjects, building a spoken language interface for programming without placing restrictions on the language that can be used would be more complex.

However, it is unrealistic to expect that all the subjects would completely share a single vocabulary. Each programmer uses a variety of novel words within their

source code, such as class/method/variable names and text within comments, that are not words that a natural language interface for programming would have to understand and process. Therefore, it is unrealistic to expect that a dictionary will ever attain complete coverage of the words uttered by users.

However, one would expect that the percentage of novel words of this nature (e.g., variable names, method names, etc.) uttered by a user should be relatively constant across users, and be a relatively small percentage of the words uttered. If this is the case, then a system should not expect to achieve 100% vocabulary coverage for new users but should expect to achieve good vocabulary coverage, since most words do represent general programming and editing command directives that should be common across users.

I conducted a set of experiments to examine the proportion of novel words used by a "new user". In these experiments, I imagined that a system has been created to support the collective vocabulary used by N subjects. I then measured the vocabulary coverage that this imaginary system would provide for the $N + 1th$ subject.

In the first experiment, I treated each subject as a new user. I combined the vocabularies of all of the other subjects into a "base" vocabulary. I then compared the new user's vocabulary against this base vocabulary to determine the percentage of unique words uttered by a "new user" that would be covered by the base vocabulary. The results are shown in Table 5.8. The first column in this table shows the number of unique words used by each subject. The second column shows the percentage of each subject's vocabulary that is covered by the base vocabulary. These results show a surprisingly high level of coverage of a new user's vocabulary, averaging 77.6%.

It is possible that the range of vocabulary coverage could, in part, result from some subjects speaking their thoughts aloud. For instance, the subject with the highest vocabulary coverage, U52, never spoke extraneous thoughts aloud. On the other hand, the subject with the lowest vocabulary coverage, U45, spoke his thoughts on a continuing basis. Unfortunately, determining the impact of spoken

Table 5.8. Percentage of each subject's vocabulary that overlaps with a vocabulary established by the other seven subjects.

Subject	Number of unique words	Percent overlap against other Subjects
U08	406	81.0
U20	572	80.9
U24	859	69.6
U39	682	80.8
U45	964	69.4
U52	426	88.3
U66	629	77.9
U96	677	72.8
Average across all subjects	651.9	77.6

thoughts on vocabulary coverage would require marking every instance of a spoken thought in the data set. This work is beyond the scope of this thesis.

Seven subjects is a small sample from which to build a core vocabulary, which makes 77.6% coverage even more impressive. But this small sample also raises the question: how quickly/slowly does vocabulary coverage grow, and at what point (if any) does vocabulary coverage level off? To examine the growth rate, I plotted a curve showing how vocabulary coverage grows as more users are added to the pool. Here's the specific procedure that I used, where S = the set of subjects, and $N = |S|$.

For $i = 1$ to $N - 1$ (where N = the number of subjects)

1. For each subject s in S
   (a) randomly select i subjects from $S - s$ and combine their vocabularies to form the base vocabulary.
   (b) compare s's vocabulary against the base vocabulary and determine the percentage of s's unique words covered by the base vocabulary.

Table 5.9. The growth of vocabulary coverage as more subjects are added. The number in the left column represents the number of other subjects used to build the base vocabulary.

Number of Subjects Used to Form the Base Vocabulary	Average Base Vocabulary Size	Average Overlap (%) for new user	Minimum Overlap (%) for new user	Maximum Overlap (%) for new user
1	607.3	45.8	29.8	55.0
2	986.5	59.7	46.8	70.9
3	1178.2	65.9	55.3	78.2
4	1484.3	71.2	61.7	83.1
5	1721.7	74.0	63.4	86.6
6	1917.1	76.0	66.7	87.6
7	2064.2	77.6	69.4	88.3

2. Average the percentages from part (b) over all subjects.

This procedure computes the average vocabulary coverage for a new user, given that the vocabularies from i other users were combined to form the base vocabulary. The averages for all values of i are shown in Table 5.9 and plotted in Figure 5.1. These results show vocabulary coverage rising fairly rapidly until the coverage attains more than 70% at four subjects, then the rate of coverage increase slows. Vocabulary coverage reaches an average of 77.6% when a new user is compared against all other subjects. These results show that good vocabulary coverage can be attained with a small number of randomly selected users. These results suggest that a dictionary can be constructed from a relatively small number of sample users that is likely to have reasonably good coverage for new users.

Comments placed within the source code represent a possible source of words that may be unique to a single subject. These words are the user's description of the code and its properties, but are not necessarily the same words that are used to create the source code. Since the words within the comments are placed, unprocessed, into the evolving program, a natural language interface would not need to take any actions based on these words. To test the impact of the words used in comments, I removed all utterances that requested a comment. This

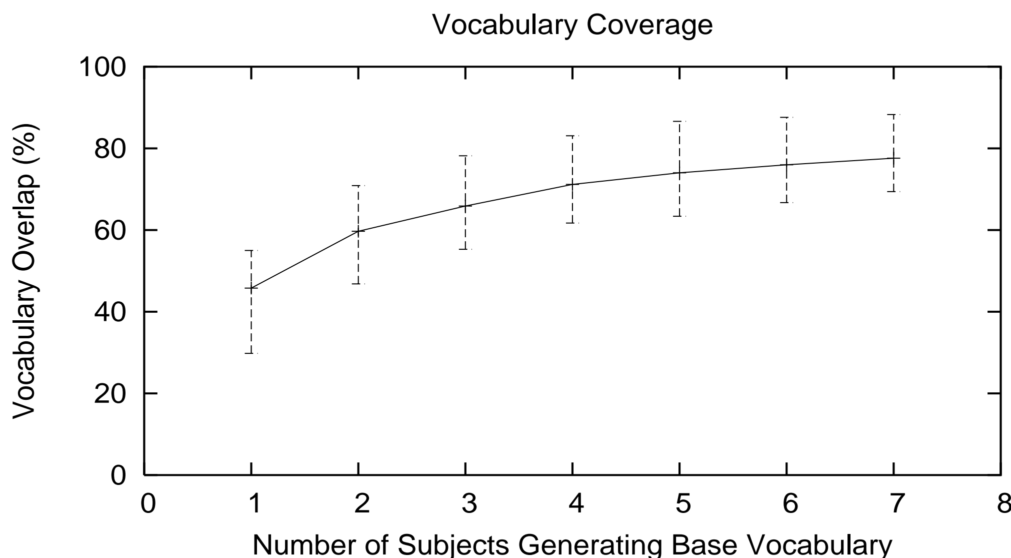

Figure 5.1. The growth of a shared vocabulary. The line shows the average overlap in vocabulary for each of the subjects when compared against a vocabulary generated by a given number of other subjects (on the x-axis). Error bars show the range in overlap for the individual subjects.

removed 738 utterances (8.2% of all utterances) from the data set. Since 62.8% of the utterances requesting a comment contained other command types (see Table 5.7), this resulted in removing 463 requests containing at least one other command type (561 additional command tags removed). Edit and navigation commands comprised 86% of these additional command tags. Table 5.10 shows the changes in the vocabulary that resulted from removing these utterances from the data set.

To test whether source code comments influenced the rate of vocabulary growth, I repeated the previous experiment, utilizing the data set containing no source code comments. The results are shown in Table 5.11 and plotted in Figure 5.2. These results show that removing the utterances including source code comments produces almost no change to the underlying trends. However, there exists a possibility that the words from the other command types removed with the comments could be influencing this distribution.

Another question that arises is how many of the word *instances* uttered by a

Table 5.10. Statistics for the data set for C++ subjects. The utterances excluded from the data set contain source code comments.

	Number of Unique Words	Number of Word Instances
All utterances	2220	65122
All utterances excluding comments	1920	57198

Table 5.11. The growth of vocabulary coverage as more subjects are added. The number in the left column represents the number of other subjects used to form the base vocabulary. All utterances containing source code comments were removed before comparison.

Number of Subjects in Base Vocabulary	Average Base Vocabulary Size	Average Overlap (%)	Minimum Overlap (%)	Maximum Overlap (%)
1	500.3	46.1	28.6	56.3
2	813.1	59.0	42.3	71.6
3	994.3	66.3	52.0	79.2
4	1319.0	72.1	63.0	83.7
5	1508.1	74.6	62.0	87.9
6	1648.6	75.8	65.3	88.5
7	1782.7	77.4	67.3	88.8

new user would be covered by the base vocabulary. If a new user's most frequently uttered words fail to exist in the base vocabulary, a spoken language interface would be unusable. Alternatively, if all but a few word instances uttered by a user existed within the known vocabulary, little frustration would result from unknown words. To examine this question, I repeated the procedure just described (see page 61). However, in step b., I determined the percentage of the user's word instances covered by the base vocabulary, rather than unique words. The results are shown in Table 5.12 and plotted in Figure 5.3.

These results show a rapid rise in instance coverage until attaining 95% coverage. After this level of coverage is attained, growth of coverage slows. Therefore, a vocabulary generated from only seven users covers the bulk of the words uttered by

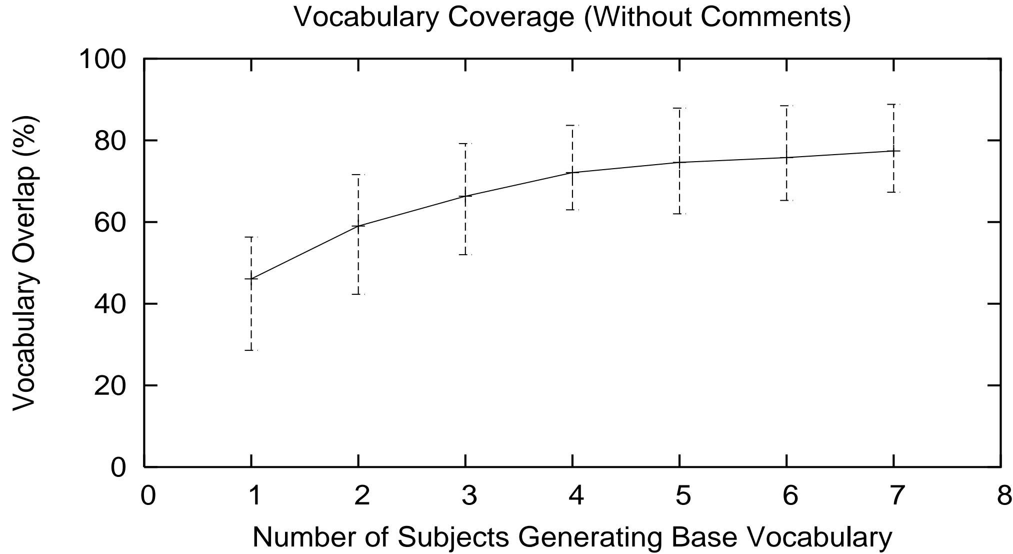

Figure 5.2. The growth of a common vocabulary when utterances containing source code comments are removed. The line shows the average overlap in vocabulary for each of the subjects when compared against a vocabulary generated by a given number of other subjects. Error bars show the range in overlap for the individual subjects.

a new user. This suggests that new users will not be frustrated because a vocabulary generated from other users lacks words crucial to them.

In summary, the subjects of the Wizard of Oz study phrased their requests in a wide variety of ways. However, the acquisition of a common vocabulary occurs quickly, reaching a vocabulary coverage of unique words approaching 80% when the base vocabulary is built from seven subjects. Furthermore, the words in the base vocabulary cover more than 95% of the word instances uttered. Given that every programmer is going to use their own novel words for variable and method names, and in comments, these high levels of vocabulary coverage suggest that a common vocabulary can be developed for a spoken language interface for programming that will have good coverage for new users of the system. However, the selection of all the subjects from a single class may introduce a bias to these results. The

Table 5.12. The growth of word instance coverage as more subjects are added. The number in the left column represents the number of other subjects used to build the base vocabulary.

Number of Subjects in Base Vocabulary	Average Base Vocabulary Size	Average Instance Overlap (%)	Minimum Instance Overlap (%)	Maximum Instance Overlap (%)
1	607.3	79.3	57.3	86.7
2	986.5	88.4	79.7	94.9
3	1178.2	92.3	89.6	96.1
4	1484.3	94.1	91.2	97.4
5	1721.7	95.0	93.0	97.6
6	1917.1	95.3	92.7	97.7
7	2064.2	95.7	93.7	98.0

subjects may have unconsciously adopted the instructor's vocabulary and phrasing to describe programming constructs.

5.4 Disfluencies

The commands uttered by the users of a spoken language interface for programming would likely contain disfluencies. These disfluencies would complicate the task of extracting information from the input in order to create the source code. However, as noted in Section 2.4, the nature of disfluencies varies across domains, so I wanted to study how disfluencies presented themselves in a natural language programming scenario. First, I will describe how the data were collected. Second, I will examine the types and frequencies of disfluencies in the programming request domain. Third, I will examine the effects these disfluencies have on an information extraction system.

5.4.1 Data Preparation for Disfluencies

To create the data set for examining the effect of disfluencies, I randomly sampled 10% of each subject's utterances from the original transcripts. I marked each instance of the four types of disfluencies described in Section 2.4: filled pauses, repetitions, false starts, and repairs. The disfluency tags indicate the

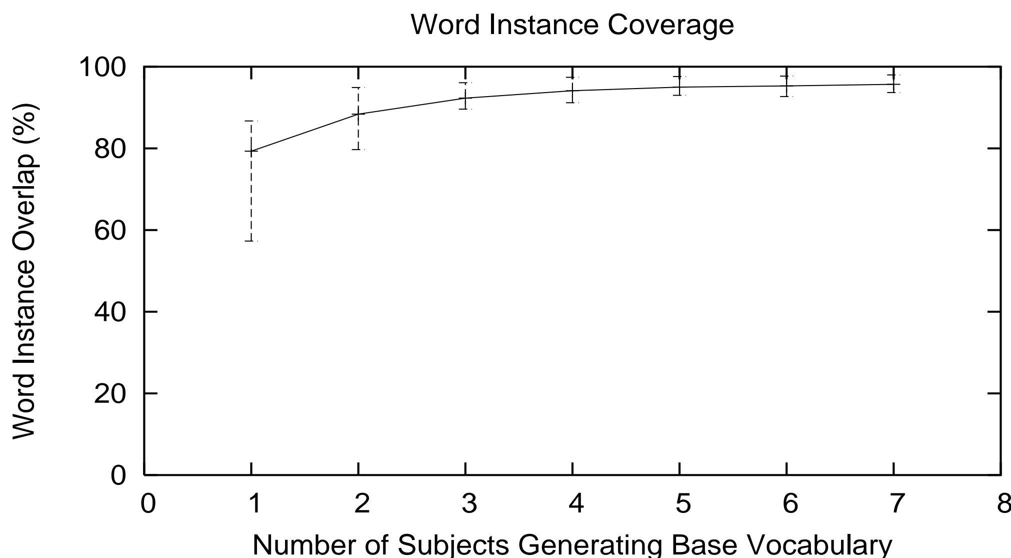

Figure 5.3. The growth of coverage in word instances. The line shows the average coverage of word instances for each of the subjects when compared against a vocabulary generated by a given number of other subjects. Error bars show the range of coverage for the individual subjects.

interruption point, the onset of the repair, and the regions being repaired in the case of repetitions and repairs. Since I instructed the subjects not to dictate punctuation, the transcripts show no indications of boundaries between different phrases, clauses, and sentences. Therefore, if the utterance contained a coherent sequence of thoughts and I could discern a reasonable grammatical structure, I assumed that there were no disfluencies.

I made one change to the data during this process. If the wizard needed to request more information from the subject using the message region (see Section 4.1), I divided the subject's original request and each response to a message into separate utterances.

5.4.2 Disfluencies in Programming Requests

It is important to understand the frequencies and types of disfluencies in the programming domain. If disfluencies pose a problem, this knowledge will be useful

in determining how best to approach this problem. I began by determining the frequencies of disfluencies for each of the subjects.

Table 5.13 shows the number of filled disfluencies committed by each subject per utterance. Filled disfluencies represent those utterances that contain a portion of the utterance that must be removed to achieve the intended utterance. This category includes filled pauses, false starts, repetitions, and repairs (see Section 2.4). Three subjects (U08, U39, and U52) have fewer than one disfluency in six requests. Subjects U20, U24, U66, and U96 span the range from one disfluency in every three utterances to a disfluency in every utterance. Subject U45 averages 1.5 disfluencies in each utterance. This high rate of disfluencies results from this subject's propensity to think aloud. While many of the subjects thought aloud occasionally, subject U45 spoke his thoughts on a continuing basis.

Table 5.14 shows the number of each type of disfluency produced by the subjects. The same three groupings of the subjects appears in this table, despite the lack of normalization. In fact, subject U45 produces 34% of all of the disfluencies.

Any disfluencies remaining in the utterance could interfere with the process of information extraction. For instance, filled pauses may be interpreted as a noun phrase, disrupting the parsing of the input and making information extraction difficult. A knowledge of the distribution of the different types of disfluencies would

Table 5.13. Rates of disfluencies across subjects. The right-hand column shows the number of filled disfluencies that occur per subject request.

Subject	Filled DFs
U45	1.50
U96	1.05
U66	0.85
U20	0.45
U24	0.34
U39	0.16
U08	0.15
U52	0.01
All Subjects	0.49

Table 5.14. The number of disfluencies, listed by type, contained within a subject's requests. Column headings are: FP = Filled Pauses; FS = False Starts; RPT = Repetitions; RPR = Repairs.

Subject	FP	FS	RPT	RPR	Total
U45	131	6	10	16	163
U96	60	3	8	20	91
U66	59	5	3	7	74
U20	13	12	20	28	73
U24	13	10	7	10	40
U39	11	1	1	8	21
U08	1	5	1	9	16
U52	0	0	0	1	1
All Subjects	288	42	50	99	479

be needed for solving this problem.

Table 5.15 shows the distribution of the different types of filled disfluencies for each subject. The majority of the subjects generate more filled pauses than any other type of disfluency, followed by a smaller number of repairs, with few false starts and repetitions, although the relationships between these values vary. For instance, Subject U20 repairs many utterances and corrects many repetitions, but utters comparatively few filled pauses or false starts. Thus, the types of filled

Table 5.15. Distribution of filled disfluencies across types.

Subject	Filled Pauses	False Starts	Repetitions	Repairs
U08	0.06	0.31	0.06	0.56
U20	0.18	0.16	0.27	0.38
U24	0.32	0.25	0.18	0.25
U39	0.52	0.05	0.05	0.38
U45	0.80	0.04	0.06	0.10
U52	0.00	0.00	0.00	1.00
U66	0.80	0.07	0.04	0.09
U96	0.66	0.03	0.09	0.22
All Subjects	0.60	0.09	0.10	0.21

disfluencies found depends strongly on the individual subject.

In summary, the nature of disfluencies within this domain varies widely among the users. While few disfluencies are present within the utterances of some subjects, they are plentiful in the utterances of others. Clearly, a spoken language interface must cope with an abundance of disfluencies. One solution is to require users to manually correct disfluencies before utterances are processed. However, users might find it cumbersome to correct disfluencies if they occur in a majority of the utterances. While manually correcting disfluencies may "train" users to plan their utterances more carefully, a spoken language interface for programming cannot rely on this training to eliminate disfluencies.

If the disfluencies are not corrected by the user, they must be corrected or ignored during the processing of the request. In the next section, I will examine the effects of disfluencies on the underlying NLP system used in NaturalJava.

5.4.3 Disfluency Impact on NLP

Information extraction is one approach to natural language processing. It seeks to find relevant information for a task. One technique for performing information extraction begins by parsing sentences to determine their syntactic structure. Since disfluencies may disrupt the syntax of a sentence, they may interfere with the process of information extraction. To test the impact of disfluencies on information extraction, I examined the effects of disfluencies on the Sundance information extraction system (see Section 3.1.1).

I used two data sets to perform this test. For the first data set, I used the original sentences that I tagged for disfluencies. To build the second data set, I utilized the disfluency tags described in section 5.4.1 to remove the reparandums from the original utterances, obtaining the effective utterances (see Section 2.4). These two data sets provided me with lists of original utterances (i.e., potentially containing disfluencies) and their corresponding effective utterances (i.e., with all disfluencies removed). Remember that, on average, half of these utterances contained disfluencies (see Table 5.13).

To determine the effect of disfluencies on Sundance's ability to extract information, I processed each of the original utterances and its corresponding effective utterance using Sundance. For each utterance, Sundance instantiates a collection of case frames. I compared the case frames instantiated for the original utterance against those for the effective utterance. The differences between these case frames resulted from the disfluencies found within the original utterance. For purposes of this comparison, I assumed that the case frames instantiated by Sundance for the effective utterances represented perfect extraction of information from the utterances. While this assumption is demonstrably false, it represents a satisfactory base case because I am measuring the impact of disfluencies on the current set of extractions. If Sundance's IE engine improves and more extractions occur, I would expect the relative impact of disfluencies to be the same.

Each case frame instantiated by Sundance contains a great deal of information. As I described in Section 3.1.1, a case frame contains four components: the word triggering the case frame, the type of the case frame, the slot names, and the strings extracted from the input (see Figures 3.6 and 3.7 for examples of case frames). Each of these components provides valuable information. The triggering word can be a useful keyword indicator while processing case frames. For instance, in NaturalJava, the trigger word is used to aid in disambiguating the *multi_purpose* case frames. The case frame type represents the concept embodied in the case frame. The slot names for the extracted strings indicate the role played by the phrase extracted in that slot. For instance, in Figure 3.6, the prepositional phrase starting with the preposition "from" indicates the initial condition of the loop and its slot name, *loop_start*, indicates that role. The extracted strings are usually noun phrases, with the most important element being the head noun. For example, given the sentence fragment "create a public class", Sundance instantiates the case frame seen in Figure 5.4. Sundance extracts the direct object noun phrase "a public class". The head noun for this phrase is "class." It is the most important element of this string because it indicates the type of object to be created.[2]

[2]The word "public" also contains important information, but this information is applied to

```
Trigger: create
Type: create
    create_type: "a public class"
```

Figure 5.4. An example of an instantiated Sundance case frame triggered by the verb "create".

I used these four components of the case frames to compare the case frames instantiated from the original utterance to those instantiated from the effective utterance. The disruptions caused by disfluencies can impact these case frame components in a variety of ways. In some cases, minor changes can occur to the data held within the case frame, but most or all of the essential data remains to carry out the action indicated by the case frame. In other cases, the intent of the case frame can still be recognized, but the disfluencies have disrupted the sentence enough so that the action can no longer be understood. I utilized three levels of equivalence between case frames to examine these differences in the impact of disfluencies on information extraction. These levels of equivalence are:

- *Exact Match*

  An exact match between case frames occurs when the trigger word, the type, all of the slot names, and all of their extracted strings are identical in both case frames. An exact match means that any disfluencies within the original utterance have not degraded Sundance's ability to extract information at all.

- *Head Noun Match*

  A head noun match indicates that the case frame type and all of the slot names are identical, and the extracted strings contain the same head nouns between corresponding case frames. However, the noun phrase modifiers in the extracted string may be different. Head Noun matches allow for the disfluency to cause a minor change to one or more of the extracted strings, but

the class (i.e., the class is public), not to the verb (i.e., a class is created, not a public is created).

the most crucial information extracted by the case frame remains the same. Figure 5.5 shows an example of two case frames instantiated by Sundance which satisfy a head noun match. Case frame A results from the original utterance, "Create a um public class", while case frame B is instantiated from the corresponding effective utterance. Both of these case frames contain the same trigger word, type, and slot name. The head noun, "class", is the same within both extracted strings, but a difference in the noun modifiers disqualifies these two case frames from being an exact match.

- *Slot Name Match*

  A slot name match indicates that the trigger word, the type and all slot names are identical between the two case frames. Slot name matches indicate that the basic concept embodied in the case frame remains the same, but the extracted information may have changed. In this case, the disfluency has a serious impact on the information extracted, but some useful information is still retained. Consider the original utterance "Create a um let's see public

```
A.)  Original Utterance Case Frame

     trigger: create
     type: create
         create_type: "a um public class"


B.)  Effective Utterance Case Frame

     trigger: create
     type: create
         create_type: "a public class"
```

Figure 5.5. A pair of case frames that satisfy the head noun match criteria. The original utterance to produce case frame A is "create a um public class".

class". The case frames instantiated by Sundance for the original and effective utterances are shown in Figure 5.6. The disfluency in the original utterance prevents the direct object noun phrase from being parsed correctly. As a result, Sundance is unable to extract the string "public class".

Table 5.16 shows the differences in Sundance's ability to extract information from the original utterances and the corresponding effective utterances. In this table and the following discussion, the term "original utterance case frames" indicates case frames generated by Sundance extracting information from the original utterances. Similarly, the term "effective utterance case frames" applies to the case frames generated from Sundance's processing of the effective utterances.

The presence of disfluencies increases the number of case frames extracted from the original utterances by 112 over the number extracted from the effective utterances. This is a surprisingly small number of additional case frames to be produced, given the presence of 479 disfluencies (see Table 5.14). However, these additional case frames do not represent the complete effect of the disfluencies.

```
A.)  Original Utterance Case Frame

     trigger: create
     type: create
         create_type: "a um"


B.)  Effective Utterance Case Frame

     trigger: create
     type: create
         create_type: "a public class"
```

Figure 5.6. A pair of case frames that satisfy the slot name match criteria. The original utterance to produce case frame A is "create a um let's see public class".

Table 5.16. Statistics on the case frames generated by Sundance and the numbers of case frames that match for both the original utterance and the effective utterance. Numbers for case frame matching are cumulative.

Total number of original utterance case frames	1,749
Total number of effective utterance case frames	1,637
Number of exact matches	1,542
Number of exact + head noun matches	1,576
Number of exact + head noun + slot name matches	1,616
Number of unmatched original utterance case frames	133
Number of unmatched effective utterance case frames	21

Slightly more than 88% of the case frames generated from the original utterances (1,542/1,749) exactly match those generated from the effective utterances. Thus, Sundance successfully extracted the correct information nearly 90% of the time despite the presence of disfluencies.

As described above, head noun matches represent situations where the disfluencies resulted in minor changes to the extracted strings within a case frame. The concept embodied by the case frame and all of the crucial information needed for that concept still remain, but the additional words of the disfluency may have been added to the extracted string. In an application like NaturalJava, these changes may cause errors. If the extracted string is used in its entirety, such as in declaring a literal string or commenting the source code, then the disfluency will appear in the source code. In other cases, relevant features are selected from the extracted strings and extraneous material is ignored. For instance, in Figure 5.5, part A, the words “public” and “class” would be selected and the remaining words discarded. Slightly less than 2% of the case frames generated from the original utterances (34/1,749) do not match exactly but satisfy the head noun match criteria. So, for 90.1% of the utterances (1,576/1,749), the disfluencies had no effect or only very minor effects on the natural language processing.

If the disfluencies result in more substantial errors in information extraction,

crucial information is lost. This is particularly true when the head noun of a phrase is not extracted properly. If the correct head noun is lost, the concept embodied by the case frame remains, but crucial information for that concept is missing. In applications like NaturalJava, loss of the head noun can result in inappropriate actions. For instance, if the case frame embodied the concept of naming an element of the source code, changing the head noun would result in changing the name of the object. In other cases, such as creating an element of the source code, the effects are much less serious. In this instance, there are a limited number of types of elements which can be created. If the head noun was not one of the known types, the user would be prompted to correct the error. Comparing original utterance case frames to effective utterance case frames yields matches to slot names for 2.3% of the original utterance case frames (40/1,749). Slot name matches increases the coverage of original utterance case frames containing relevant information to 92.4% (1,616/1,749).

The 7.6% of the original utterance case frames that do not match to effective utterance case frames pose a potential problem for any future spoken language interface for programming. Disfluencies within the original utterances resulted in the extraction of extraneous information. Any future interface would need the ability to identify and discard this extraneous information.

However, the case frames instantiated from the effective utterances that do not match to a case frame found within the original utterances demonstrate a more serious problem. These unmatched effective utterance case frames comprise 1.3% of all effective case frames. These unmatched effective utterance case frames indicate that the syntax of the utterance was disrupted enough by the disfluency to prevent Sundance from extracting any information at all.

In summary, disfluencies present problems for a spoken language interface for programming. However, an information extraction system appears to be surprisingly resilient in the presence of these disfluencies. Sundance extracts approximately 90% of the information required for processing user requests. Therefore, the problems posed by disfluencies do not appear to be catastrophic for a spoken language

interface for programming. However, Sundance is currently unable to extract all relevant information from the user requests. Therefore, these results are shown with respect to Sundance's current ability to extract information. As Sundance's capabilities improve, these results may change. Additionally, these results may also be skewed because I sampled only 10% of the total data set to create the data used for this study. Errors generated by a speech recognition system may also interfere with information extraction, but issues associated with speech recognition lie outside the scope of this study.

5.5 Informal Feedback

At the end of each session, I asked the subjects to fill out a questionnaire about their experience using the interface. I asked them to answer four open-ended questions:

- *Did you find the natural language programming system easy to use?*
- *What features worked best?*
- *What features did not work well?*
- *What suggestions do you have to make this system easier to use?*

I asked these questions in order to ascertain how well the subjects liked using a spoken language interface for programming and what features they would like to have in such an interface.

The reaction from the subjects to using a spoken language interface for programming was overwhelmingly positive. They found the interface easy to use in most ways. Many liked the assistance the interface provided with syntax; they said that they knew what they wanted the statement to do, but they could not remember the exact syntax. Their complaints about the system revolved around its slow speed of processing and problems that we occasionally had with our testing infrastructure.

The subjects expressed many opinions about the features they desire in a spoken language interface for programming. Essentially, they would like a multi-modal user interface with most of the features of a standard integrated development environment. I combined their comments with my own observations to develop the following list of suggested elements for a spoken language interface:

- *Push to talk switch*

  While two of the subjects uttered few extraneous words, most subjects spoke their thoughts aloud from time to time. In fact, one subject thought aloud during most of his time during the study. These additional words make the task of determining the types of commands present within the request more difficult and complicate the task of information extraction. Adding a push to talk switch allows the users to think aloud if they desire while providing better input for the interface.

- *Integrated compilation and debugging support*

  Compilation and debugging must be integrated into the user interface for ease of use. This is particularly important for persons with mobility impairments, who would have difficulties when forced to leave the spoken language interface to compile and debug their source code.

- *Multiple, resizable windows*

  Many of the subjects desired to display several files simultaneously.

- *A mouse*

  A mouse should be integrated into the environment to add flexibility in navigation and selection tasks. For instance, the subjects found navigation to known locations within the source code in the spoken language environment to be easy, but scanning through the code or finding new locations within the source code was cumbersome.

- *A keyboard*

  A keyboard should be integrated to quickly correct errors generated by the

speech recognition system. These problems include homophones (e.g., hear vs. here) and numbers (twelve vs. 12).

- *Syntax Highlighting*

  The novice programmers who took part in the Wizard of Oz study stated that they found their source code much easier to interpret when it was color-coded to indicate syntax. Syntax highlighting also assists persons with vision loss who use speech synthesizers as their output device because pitch changes used to indicate syntactic elements in the source code improve listening comprehension.

- *Line numbers displayed alongside source code*

  Many subjects found it cumbersome to navigate within the source code displayed in the current window—it often required counting lines up or down from the cursor position. Line numbers adjacent to the source code would make moving to these locations easier.

- *Error messages detailing why processing of speech input failed*

  When subjects made requests that the wizard could not understand, the subjects often received "Command not understood" as the error message. Many subjects found this lack of information frustrating, wanting to know if they did not use the correct words or if they mumbled. These subjects wanted to know if they should rephrase the request or just state the request more clearly. Thus, error messages stating why the user's request could not be processed are important.

- *Extensive help files*

  The subjects for this study were given little direction about how to use the interface so that we did not bias how they used it. Many subjects requested help files with examples to show them how certain types of requests might be made.

Some of the features described above are not applicable for all types of users. For instance, multiple windows for viewing source code would be of limited utility for persons with vision loss. Other features, however, would need to be implemented in a manner that would allow utility for all users. For instance, persons with vision loss would desire the ability to control the mouse using the keyboard's numeric keypad.

5.6 Chapter Summary

Many difficult problems remain to be solved before a flexible, fully-functional spoken language interface for programming can be developed. The goal of this research is to provide an understanding of a few of the basic issues involved in this development process. I investigated three specific questions relevant to a spoken language interface for programming. These questions, and the associated results, are listed below.

- What types of commands did the students use and how did they use them?
  The command types contained within requests from the subjects were generally simple. Requests contained only one or two command types more than 90% of the time, and subjects primarily used a small subset of the possible command types.

- Do the students share a common vocabulary or do the words they use differ substantially from student to student?
  The subjects phrased their requests using a wide variety of vocabulary. However, a common vocabulary could be built using a small number of subjects. A vocabulary built from seven subjects covered nearly 80% of the unique words used by a "new user", and this vocabulary covered more than 95% of the words uttered.

- What types of disfluencies are used by the students, how frequently do these disfluencies occur, and how do they impact the underlying natural language processing system that extracts information from the user requests?

> Disfluencies occurred, on average, in 50% of the requests in the programming domain. Filled pauses comprised the bulk of the disfluencies but the types and frequencies of disfluencies varied widely across the subjects. However, the Sundance information extraction system performed surprisingly well in the presence of the disfluencies, extracting nearly 90% of the information without error.

These results are limited in a number of ways. First, all of the subjects were novice programmers. Second, all of the subjects were students in a single introductory programming class. The instructor may have influenced how they described programming constructs through his use of language during class. Third, this study covers a small number of subjects performing a relatively small number of tasks. As a result, the data collected during this study may not represent a general population. Finally, the results indicating Sundance's ability to extract information despite the presence of disfluencies are shown with respect to Sundance's current ability to extract information. As Sundance's capabilities improve, these results may change.

I utilized feedback from the subjects and my own observations to develop a list of suggested features for a spoken language interface for programming. These features should be added to the features offered by an integrated development environment. The additional features include a push-to-talk switch, integrating speech input with mouse and keyboard input, line numbers to ease spoken navigation, and specific error messages regarding failures when processing requests.

Additionally, the reaction from the subjects to using a spoken language interface for programming was overwhelmingly positive. In particular, they appreciated the assistance with syntax provided by the interface. This result suggests that novice programmers would use a spoken language interface for programming.

CHAPTER 6

CONCLUSIONS

A spoken language interface for programming represents one possible solution to the problems posed by programming language syntax. The complexities of the syntax pose problems for many groups of potential programmers. These groups include novice programmers, persons with vision or mobility impairments, and advanced programmers using a new programming language.

Many difficult problems remain to be solved before a flexible, fully-functional spoken language interface for programming can be developed. The goal of this research is to provide an understanding of a few of the basic issues faced in this development process. I investigated three aspects of a spoken language interface for programming in the context of novice programmers taking an introductory programming class. These issues are summarized in the following research questions:

- What types of commands do the people use and how do they use them?
- Do the people share a common vocabulary or do the words they use differ substantially from person to person?
- What types of disfluencies are used by the people, how frequently do these disfluencies occur, and how do they impact the underlying natural language processing (NLP) system that extracts information from the user requests?

The results addressing these three questions are summarized in the following paragraphs.

The complexity of the requests made by the users will be a factor in any future interface. This complexity appears in the number of command types, or actions,

contained within each request. Increasing the number of command types within a request increases the difficulty of determining the actions being requested. In general, the subjects in this study made simple requests of the interface. Most of the requests contained only one or two command types. Additionally, the bulk of the subject requests utilized a small subset of the possible command types. These results show that the subjects made relatively simple requests of the interface.

A spoken language interface must be able to understand the language comprising the requests. If all of the subjects use a consistent vocabulary, future systems can be designed around this common vocabulary. If each user utilizes a different vocabulary, building a spoken language interface for programming without placing restrictions on the language that can be used would be more complex. In this study, the subjects phrased their requests using a wide variety of vocabulary. However, a common vocabulary built from seven subjects covers nearly 80% of the unique words used by a “new user”. The unique words in this common vocabulary cover more than 95% of the words uttered by the new user. These results suggest that a common vocabulary can be generated from a relatively small number of users.

Disfluencies are parts of an utterance that must be removed in order to achieve the speaker’s intended utterance. They are artifacts of human speech that impact spoken language interfaces. In particular, disfluencies disrupt the syntax of a sentence, which can impact some methods of information extraction. In this study, subjects averaged a disfluency in every second utterance, although the rates and types of disfluencies varied widely across the subjects. However, the Sundance information extraction system performed surprisingly well despite the disfluencies. Sundance extracted nearly 90% of the information without error. Thus, some methods of information extraction appear to be remarkably resilient despite the presence of disfluencies.

One aspect of this study may limit the application of these results. All of the subjects for this study were recruited from a single introductory programming class. As a result, all of the subjects taking part in the study were novice programmers, so these results may not generalize to other populations of programmers. Additionally,

since all of the subjects were learning programming skills from the same instructor, the instructor's choice of vocabulary and phrasing for programming constructs may have been adopted by the subjects. Finally, this study observed a small number of subjects working on a relatively small number of tasks. Therefore, these results may not apply to a general population working on a wide variety of tasks.

Another issue that may impact the reliability of these results pertains to the apparently robust performance of the Sundance information extraction system in the presence of disfluencies. Sundance is currently unable to extract all relevant information from the user requests. Therefore, the results indicating Sundance's ability to extract information despite the presence of disfluencies are shown with respect to Sundance's current ability to extract information. As Sundance's capabilities improve, these results may change.

The Wizard of Oz Study also allowed me to observe the subjects interacting with a spoken language interface for programming. I used these observations and informal feedback from the subjects to develop a list of suggested features for a spoken language interface for programming. Essentially, the interface should contain all of the features of a standard integrated development environment. Additional features should be integrated into this environment. These features include a push-to-talk switch, integrating speech input with mouse and keyboard input, line numbers to ease spoken navigation, and specific error messages regarding failures when processing requests.

The opportunity to observe the subjects interacting with a spoken language interface for programming also allowed me to evaluate their reaction to using such an interface. The subjects of the study were enthusiastic about using this type of interface for programming. They particularly enjoyed the assistance with syntax provided by the interface. Therefore, I believe that a spoken language interface would provide a useful tool for learning to program.

Additional research could also be carried out using the data collected during this study. For instance, the method for tagging user requests with the command types they contain could be changed. I chose the method used in this study to

avoid biasing the tagging to complement the Sundance natural language processing system. However, embedded XML-style tags could be used to encapsulate the portion of each utterance relevant to a given command type. This style of tagging would allow other research questions to be addressed. For instance, the vocabulary associated with any given command type could be collected, and overlapping vocabularies between command types could be defined. Another issue that could be studied relates to the boundaries between command types: do pairs of command types tend to be nested, overlapping, or well-separated, and are there grammatical boundaries, such as phrase or clause boundaries, that tend to divide command types. Additionally, these embedded XML-style command tags would allow better definition of the complexities of the user requests containing multiple instances of the same command type within a single request.

One issue not addressed in this research is the potential interactions between the different components of a spoken language interface for programming. Errors occurring in one component of the spoken language interface could generate additional errors in other components of the system. For instance, if a crucial word does not exist in the vocabulary of the system, then the types of commands present within the request may not be properly classified. If the types of commands contained within a user's request are not properly classified, then incorrect actions would be taken when extracting information from the request and attempting to generate source code. While the additional errors generated from the interactions between different aspects of a system will pose problems, studying these interactions is beyond the scope of this research.

Many difficult problems remain to be solved before a flexible, natural spoken language interface for programming can be developed. These problems span many fields, from speech recognition to information extraction to utilizing the extracted information to create and modify source code. For instance, many phrases used to describe computer programs (e.g., "for loop", "while loop", "plus plus", etc.) do not occur in normal human speech. Therefore, new models for speech recognition may need to be developed. Similarly, the unusual phrases that pose problems for

speech recognition may pose problems for natural language processing systems. As a result, these NLP systems may require modification to handle these unusual phrases. Additionally, the knowledge needed to carry out information extraction in this domain must be acquired and implemented. Finally, the wide variety of expressions that can be used to request that a single type of action be taken poses many problems. For instance, determining each individual action being requested, given the wide range of potential actions that can be taken when creating or editing source code, is just one problem remaining to be solved.

REFERENCES

[1] Biermann, A., Ballard, B., and Sigmon, A. An Experimental Study of Natural Language Programming. *International Journal of Man-Machine Studies 18* (1983), 71–87.

[2] Califf, M. *Relational Learning Techniques for Natural Language Information Extraction*. PhD thesis, The University of Texas at Austin, 1998.

[3] Cardie, C. Empirical Methods in Information Extraction. *AI Magazine 18*, 4 (1997), 65–80.

[4] Chisholm, W., Vanderheiden, G., and Jacobs, I. Web Content Accessibility Guidelines, 1.0. World Wide Web Consortium.

[5] Dahlstrom, D. A System for Wizard of Oz Studies in Natural Language Programming, Bachelor's Thesis, School of Computing, University of Utah (2001).

[6] Désilets, A., Fox, D. C., and Norton, S. VoiceCode: An Innovative Speech Interface for Programming-by-Voice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006), pp. 239–242.

[7] Freitag, D. Multistrategy Learning for Information Extraction. In *Proceedings of the 15th International Conference on Machine Learning* (1998).

[8] Freudenberger, S., Schwartz, J., and Sharir, M. Experience with the SETL Optimizer. *ACM TOPLAS 5* (1983), 26–45.

[9] Hirschman, L., Bates, M., Dahl, D., Fisher, W., Garofolo, J., Hunicke-Smith, K., Pallett, D., Pao, C., Price, P., and Rudnicky, A. Multi-Site Data Collection for a Spoken Language Corpus. In *Proc. DARPA Speech and Natural Language Workshop '92* (Harriman, New York, 1992), pp. 7–14.

[10] Jurafsky, D., and Martin, J. H. *Speech and Language Processing*. Prentice Hall, 2000s.

[11] Karadeniz, Z. I. Template Generator Using Natural Language (TEGNALAN), Bachelor's Thesis, Department of Computer Engineering, T.C Yeditepe University (2003).

[12] Lewin, I., Becket, R., Boye, J., Carter, D., Rayner, M., and Wir'en, M. Language Processing for Spoken Dialogue Systems: Is Shallow Parsing Enough. In *Accessing Information in Spoken Audio: 23 Proceedings of ESCA ETRW Workshop* (1999), pp. 37–42.

[13] Liu, H., and Lieberman, H. Programmatic Semantics for Natural Language Interfaces. In *Proceedings of CHI 2005* (2005).

[14] Maclay, H., and Osgood, C. Hesitation Phenomena in Spontaneous Human Speech. *Word 15* (1959), 19–44.

[15] Miller, P., Pane, J., Meter, G., and Vorthmann, S. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments 4*, 2 (1994), 140–158.

[16] Moore, R., and Morris, A. Experiences Collecting Genuine Spoken Enquiries Using WOZ Techniques. In *Fifth DARPA Workshop on Speech and Natural Language* (1992).

[17] Price, D., Riloff, E., Zachary, J., and Harvey, B. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 2000 International Conference on Intelligent User Interfaces* (2000), pp. 207–211.

[18] Rich, C., and Waters, R. C. Approaches to Automatic Programming. In *Advances in Computers* (1993), M. C. Yovits, Ed., Academic Press.

[19] Riloff, E. An Empirical Study of Automated Dictionary Construction for Information Extraction in Three Domains. *Artificial Intellligence 85* (1996), 101–134.

[20] Riloff, E. Automatically Generating Extraction Patterns from Untagged Texts. In *Proceedings of the 13th National Conference on Artificial Intelligence* (1996).

[21] Riloff, E. Information Extraction as a Stepping Stone toward Story Understanding. In *Computational Models of Reading and Understanding*, A. Ram, Ed. The MIT Press, 1996.

[22] Riloff, E., and Phillips, W. An Introduction to the Sundance and AutoSlog Systems. Technical Report UUCS-04-015, School of Computing, University of Utah, 2004.

[23] Shriberg, E. E. *Preliminaries to a Theory of Speech Disfluencies.* PhD thesis, The University of California at Berkeley, 1994.

[24] Soderland, S. Learning Information Extraction Rules for Semi-structured and Free Text. *Machine Learning 34* (1999), 233–272.

[25] Walker, M. A., Fromer, J., Fabbrizio, G. D., Mestel, C., and Hindle, D. What Can I Say? Evaluating a Spoken Language Interface to Email. In *CHI* (1998), pp. 582–589.

[26] Wonisch, M. *Ein objektorientierter interaktiver Interpreter fur naturalichsprachliche Programmierung"*. PhD thesis, Diploma Thesis. Lehrstuhl fur MeBtechnik, RWTH Aachen., 1995.