Book Title
Book Editors
IOS Press, 2003

A Study to Evaluate a Natural Language Interface for Computer Science Education

David E. Price, Ellen Riloff, Joseph L. Zachary
School of Computing, University of Utah, Salt Lake City, UT 84112

Abstract. Programming language syntax is a barrier to learning programming for novice programmers and persons with vision or mobility loss. A spoken language interface for programming provides one possible solution to the problems these groups face. We developed a prototype text-based natural language interface for Java programming that accepts English sentences from the keyboard and produces syntactically correct Java source code. We also conducted a Wizard of Oz study to learn how introductory programming students might ultimately use a fully functional version of a spoken language programming interface. This study shows that students typically make simple requests, that they tend to use a shared vocabulary, and that disfluencies will pose a challenge to a working system.

1. Introduction

Students learning to write computer programs are often put off by the arcane complexity of programming language syntax. Students with vision loss are unable to see syntactic presentations in the classroom, and they quickly discover that debugging syntactic errors is an inherently visual task. Students with mobility loss encounter extreme difficulty entering source code at the keyboard.

All of these students could benefit from a way to create programs without having to worry about the complexities of syntax. We are in the early stages of developing a spoken language interface for writing computer programs. Our goal is a system in which the user talks to a computer using natural English sentences or sentence fragments, in response to which the computer generates syntactically correct Java source code. We are *not* trying to create a system that can synthesize programs from high-level specifications spoken by non-programmers. In the system that we envision, students will describe the creation of programs, step by step, in much the same way that an instructor in an introductory programming class might describe program creation in a lecture (Figure 1). Our goal is to enable students to work at a higher level of abstraction by allowing them to program without getting bogged down in syntactic details.

To test whether a natural language interface for programming is feasible, we implemented a prototype that reads English requests from the keyboard and produces syntactically correct Java source code in response [8]. This prototype, *NaturalJava*, works within a small subset of Java and, more significantly, within the small subset of English that we found sufficient to describe programs.

Many technical challenges and questions must be addressed before our simple text-based language interface can be generalized into a fully functional and robust spoken

1. Create a public method called dequeue.
2. It returns a comparable.
3. Defi ne an int called index and initialize it to 1.
4. Declare an int called minIndex equal to 0.
5. Create a comparable called minValue which is equal to elements' fi rstelement cast to a comparable.
6. Loop while index is less than elements' size.

Figure 1. Sample input sentences given to NaturalJava.

language interface for programming. We investigated three questions surrounding the use of spoken requests by novice programmers to create programs: How complex are typical user requests? To what degree do the vocabularies employed by different users overlap? What is the impact of disfluent speech on information extraction? In this paper, we describe our efforts to answer these questions and present our results.

To answer these questions, we needed to observe beginning students as they used a spoken language interface for programming. In order to do this without first building such an interface, we conducted a *Wizard of Oz* study in which volunteer subjects believed that they were interacting with a working system. This bit of subterfuge was critical because studies show that subjects who believe that they are interacting with a computer behave differently than subjects who believe they are interacting with other people [7].

In our study, we asked novice programmers from an introductory C++ class to use our "system" to work on their assignments [10]. Each subject sat in front of a computer displaying what appeared to be a spoken language interface for creating programs. In reality, an expert programmer (the "wizard") was using a remote computer to monitor the subject's utterances and generate the correct responses on the subject's computer.

We recorded and analyzed all subject interactions. Our analysis shows that the individual requests that subjects made of the interface were simple in nature, that the subjects as a group used substantially the same vocabulary when making requests, and that speech disfluencies will pose a problem for working interfaces. We also found, however, that natural language processing technology performs surprisingly well despite disfluencies.

Section 2 discusses previous research in syntax-free methods of programming. Section 3 describes the NaturalJava prototype. Section 4 describes our Wizard of Oz study. Section 5 discusses the results of our analysis of the study. Finally, Section 6 presents the conclusions we drew from our research.

2. Related Work

Previous work on syntax-free programming spans three fields: automatic programming, structure editors, and natural language-based interfaces for programming. The goal for automatic programming is the generation of computer programs from an end-user's non-technical description [11]. Unfortunately, this goal is well beyond the state of the art.

Structure editors, also known as syntax-directed editors, do not allow the user to create a syntactically incorrect program [6]. Instead, each modification to the evolving source code, made by selecting from menus and filling in templates, is governed by the grammar of the programming language. Such interfaces are cumbersome for the blind.

Several natural language-based interfaces for programming have been developed. All but one of these accept input from the keyboard. NLC [1] accepts natural language

input and carries out matrix manipulations, but creates no source code. Moon [15] is an interpreted programming language that approximates natural language understanding. TUJA [4] generates frameworks for Java code (class, method, and data member declarations) from Turkish sentences. Metafor [5] builds, from the linguistic structure of English sentences, "scaffolding code" in Python to demonstrate the framework of a solution.

VoiceCode [2] is the first spoken language programming interface. It accepts spoken instructions using a constrained syntax and produces source code in one of two languages (Python or C++). VoiceCode uses templates to generate common programming constructs, such as loops and class declarations. While several specific keywords are associated with each of these templates to allow the user some flexibility in phrasing commands, the grammatical structure of the user requests is rigidly defined.

3. The NaturalJava Prototype

Because we wanted to ensure that it was possible to create Java programs using natural language we implemented a prototype that takes English sentences and sentence fragments typed on a keyboard and generates Java source code. In this section, we briefly 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. 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.

3.1. Architecture of the Prototype

The NaturalJava user interface has three components[8]. The first component is Sundance [14], a natural language processing system that uses a shallow parser to perform information extraction [13,12]. For NaturalJava, Sundance extracts information from English sentences and generates case frames representing programming 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. PRISM uses TreeFace to manage the syntax tree of the program being constructed.

Figure 2 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 components 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,

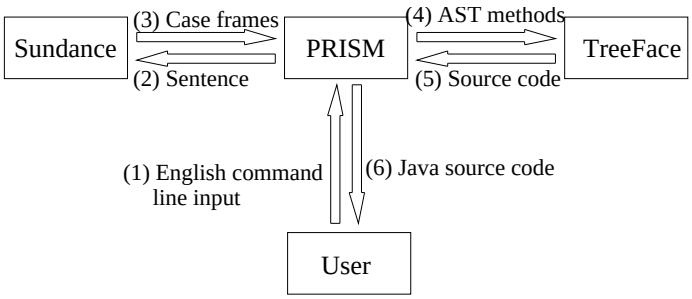

Figure 2. Architecture of the NaturalJava prototype.

TreeFace transforms the syntax tree into Java source code and makes it available to PRISM. PRISM displays this source code to the user, and can save it to a file at any time.

3.2. Limitations of the Prototype

The NaturalJava prototype possesses a number of limitations resulting from our depth-first development strategy. Its interface is very simple, allowing only input typed at the keyboard. It is best suited to program creation—editing features are extremely limited.

NaturalJava's input and output languages are limited in several ways. Its English vocabulary is limited to the words we use to describe programs. Additionally, Sundance has problems correctly parsing some of the unusual grammatical structures that might be used in programming. For output, NaturalJava supports most of the core Java language, but supports little of the Java API.

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 text-based natural language interface, albeit one with the limitations described in the previous section. To study how students would use a spoken language interface for programming, we designed and conducted a *Wizard of Oz* study[10]. We recruited eight subjects from an introductory C++ programming class to participate in this study. (We chose students from a C++ class because no Java class was offered; fortunately for our goals, Java and C++ are syntactically and conceptually similar.)

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.

During the study, each subject sat at a computer that displayed a simple one-window user interface [10]. 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 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 wizard communicated with the user by displaying source code, messages, and status updates in one of the three regions.

Before each subject's first session, we explained the purpose of the three window regions. We advised them to think of the computer as a classmate and to describe the program they wanted to that person. All source code modifications and messages were logged in coordination with the subject's audio requests. This made it possible, during the subsequent analysis phase, to recreate the exact sequence of events of each session.

5. Wizard of Oz Study Results

The goal of our research was to provide an understanding of three basic issues that arise in the development of a spoken language interface for programming: the complexity of requests made of the interface, the commonality of vocabulary across users, and the problems posed by disfluent speech on information extraction. We focused on a single potential group of users: novice programmers taking an introductory programming class.

The Wizard of Oz study comprised 67 sessions totaling approximately 125 hours. 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. We manually transcribed the audio recording of each request to make further analysis possible.

Results from our analysis of the data set make up the remainder of this section. Recall that we suggested the students talk to the computer as to a classmate. This resulted in very noisy data with many disfluencies and, in some cases, a lot of thinking aloud. In Section 5.1, we discuss the types of commands the subjects used. In Section 5.2, we analyze the vocabulary used by the subjects. The final section examines the effect that disfluencies had on the underlying NLP system used in NaturalJava.

5.1. 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, such as moving within the source code or declaring variables. We refer to each distinct type of action as a command type.

We manually labeled each request with one or more tags that indicate the types of commands that the request expresses [9]. The set of tags that characterize a request is unordered (doesn't reflect the order of command types) and contains no duplicates (doesn't reflect multiple occurrences of command types).

The relative frequencies of the different command types were *navigation* (24%), *edit* (22%), *declaration* (8%), *I/O* (7%), *parameter passing* (6%), *comment* (5%) , *method calls* (5%), *system I/O* (4%), *assignments* (4%), *other* (15%). 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.

The presence of more than one command type within a single utterance further complicates the classification of user requests. For instance, "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. Our data show that 61.0% of the commands contain only one command, 28.5% of the requests contain exactly two commands, and 7.7% of requests contain exactly three commands. Since 97% of the requests contain three or fewer types of commands, an automated procedure for classifying command types need not be overly concerned with complicated requests.

In summary, our data show that the subjects tended to make simple requests of the interface, using predominantly one or two commands within a request and using a small subset of command types for the bulk of their requests. These results suggest that determining the types of actions being requested by novice programmers is not an overly complex task. However, the small number of subjects, working on a small number of tasks, may not provide a large enough sample size to reliably generalize these results.

5.2. 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.

For the following analyses, we removed all stop words [3] from the transcripts. Throughout this section, *word instances* refers to the collection of all of the 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 these experiments, we imagined that a system has been created to support the collective vocabulary used by N subjects. We then measured the vocabulary coverage that this imaginary system would provide for the $N+1th$ subject. In the first experiment, we treated each subject as a new user. We combined the vocabularies of all of the other subjects into a "base" vocabulary. We 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 1.

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

Table 1. Percentage of each subject's vocabulary that overlaps with a vocabulary established by the other seven subjects.

These results show a surprisingly high level of coverage of a new user's vocabulary, averaging 77.6%. Furthermore, a similar analysis of word instances (instead of unique words) resulted in coverage of more than 95% of a new user's vocabulary[9].

In summary, 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. These high levels of vocabulary coverage suggest that a common vocabulary can be developed. However, the selection of all the subjects from a single class may introduce a bias to these results because the students may have adopted the language used by their instructor.

5.3. Disfluencies

Spoken language often contains disfluencies. A disfluency is a contiguous portion of an utterance that must be removed to achieve the speaker's intended utterance. There are four types of disfluencies: filled pauses, such as um, er, uh (e.g., "declare a um public class"); repetitions, where words are repeated exactly (e.g., "declare a declare a public class"); false starts, where the first part of the utterance is discarded (e.g., "declare a move down three lines"); and repairs, where aspects of the utterance are changed (e.g., "Declare a private no public class"). In each case, removing the disfluency results in the intended utterance (e.g., "Declare a um public class" becomes "declare a public class").

To create the data set for examining the effect of disfluencies, we randomly sampled 10% of each subject's utterances from the original transcripts and marked each instance of a disfluency. We found that the frequencies of disfluencies within this domain varies widely among the users but, on average, disfluencies occur in 50% of the subject requests [9]. Clearly, a spoken language interface must cope with an abundance of disfluencies.

Information extraction (IE) is one approach to natural language processing. It seeks to find relevant information for a task. One technique for performing IE 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 IE. To test the impact of disfluencies on IE, we examined the effects of disfluencies on the Sundance IE system [14].

Trigger: declare
Type: create
CreateType: "a public class"

Figure 3. An example of an instantiated Sundance case frame triggered by the verb 'declare'.

We used two data sets to perform this test. For the first data set, we used the original sentences that we tagged for disfluencies. To build the second data set, we removed the disfluencies from the original utterances, obtaining the intended utterances. These two data sets provided us with lists of original utterances (i.e., potentially containing disfluencies) and their corresponding intended utterances (i.e., with all disfluencies removed).

We processed each of the original utterances and its corresponding intended utterance using Sundance. For each utterance, Sundance instantiates a collection of case frames. We compared the case frames instantiated for the original utterance against those for the intended utterance. The differences between these case frames resulted from the disfluencies found within the original utterance.

Each case frame instantiated by Sundance 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. Each of these components provides valuable information. The triggering word can be a useful keyword indicator while processing 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. The extracted strings are usually noun phrases, with the most important element being the head noun. For example, given the sentence fragment "declare a public class", Sundance instantiates the case frame seen in Figure 3. 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.

We used these four components of the case frames to compare the case frames instantiated from the original utterance to those instantiated from the intended utterance. We utilized three levels of equivalence between case frames to examine the impact of disfluencies on IE. These levels of equivalence are:

- *Exact Match:* All components of the case frames are identical.
- *Head Noun Match:* Modifiers in the extracted string differ, but all other components of the case frames are identical. For instance, for the original utterance, "declare a um public class", the original utterance case frame would contain the extracted string "a um public class", while the intended utterance case frame would contain the extracted string "a public class".
- *Slot Name Match:* The head noun (generally right-most) is missing, but all other components of the case frames are identical. For example, the original utterance "declare a um let's see public class" results in the extracted string "a um" in the original utterance case frame, while the intended utterance case frame contains "a public class". The concept embodied by the case frame remains, but crucial information is lost.

Table 2 shows the differences in Sundance's ability to extract information from the original utterances and the corresponding intended utterances. Slightly more than 88% of the case frames generated from the original utterances exactly match those generated

	Count	Percent
Exact matches	1,542	88.2
Exact + head noun matches	1,576	90.1
Exact + head noun + slot name matches	1,616	92.4

Table 2. Statistics on the numbers of case frames that match for both the original utterance and the effective utterance.

from the intended utterances. Thus, Sundance successfully extracted the correct information 88.2% of the time despite the presence of disfluencies.

Slightly less than 2% of the case frames generated from the original utterances do not match exactly but satisfy the head noun match criteria. So, for 90.1% of the utterances, the disfluencies had no effect or only very minor effects on the natural language processing. Matches to slot names occur for 2.3% of the original utterance case frames. Slot name matches increases the coverage of original utterance case frames containing relevant information to 92.4%.

In summary, disfluencies present problems for a spoken language interface for programming. However, an IE 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.

6. Conclusions

The first step in a computer science education is learning to program. The complexities of programming language syntax pose problems for many groups of potential students. These groups include novice programmers as well as persons with vision or mobility impairments. These problems present barriers to students wishing to enter the field of computer science. A spoken language interface for programming is a potential method to remove these barriers.

Many difficult problems remain to be solved before a flexible, fully-functional spoken language interface for programming can be developed. The goal of our research was to provide an understanding of some of the basic issues faced in this development process. We investigated three aspects of a spoken language interface for programming in the context of novice programmers taking an introductory programming class. We found the following results: (1) 97% of the requests made by subjects contained three or fewer command types, suggesting that a real system will not need to deal with overly complex requests; (2) almost 80% of the vocabulary of a new user is covered by a vocabulary derived from a small number of existing users, suggesting that few restrictions will need to be placed on a user's vocabulary; and (3) disfluencies will pose a problem for a spoken language interface for programming, but information extraction technology is resilient, extracting $\approx$90% of the information correctly.

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, we believe that a spoken language interface would provide a useful tool for learning to program and break down some of the barriers keeping students from entering computer science.

7. Acknowledgments

The work described here was supported by the NSF under grants DUE-0088811, IIS-9704240, IRI-9509820, and IIS-0090100. Many thanks to Ben Newton, Dana Dahlstrom, Hung Huynh, Ryan Pratt, and Henry Longmore for their assistance with this study. 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.

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] 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.

[3] JURAFSKY, D., AND MARTIN, J. H. *Speech and Language Processing.* Prentice Hall, 2000s.

[4] KARADENIZ, Z. I. Template Generator Using Natural Language (TEG-NALAN), Bachelor's Thesis, Department of Computer Engineering, T.C Yeditepe University (2003).

[5] LIU, H., AND LIEBERMAN, H. Programmatic Semantics for Natural Language Interfaces. In *Proceedings of CHI 2005* (2005).

[6] 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.

[7] MOORE, R., AND MORRIS, A. Experiences Collecting Genuine Spoken Enquiries Using WOZ Techniques. In *Fifth DARPA Workshop on Speech and Natural Language* (1992).

[8] 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.

[9] PRICE, D. E. Using a 'Wizard of Oz' Study to Evaluate a Spoken Language Interface for Programming, Master's Thesis, School of Computing, University of Utah (2007).

[10] PRICE, D. E., DAHLSTROM, D., NEWTON, B., AND ZACHARY, J. L. Off to See the Wizard: Using a 'Wizard of Oz' Study to Learn How to Design a Spoken Language Interface for Programming. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference* (2002), pp. T2G23–29.

[11] RICH, C., AND WATERS, R. C. Approaches to Automatic Programming. In *Advances in Computers* (1993), M. C. Yovits, Ed., Academic Press.

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

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

[14] 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.

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