OFF TO SEE THE WIZARD: USING A "WIZARD OF OZ" STUDY TO LEARN HOW TO DESIGN A SPOKEN LANGUAGE INTERFACE FOR PROGRAMMING

David E. Price[1], Dan a Dahlstrom[2], Ben Newton[3], and Joseph L. Zachary[4]

Abstract *— We are in the early stages of developing a spoken language interface that will help beginners write programs. Our goal is a system in which a student will talk to a computer using English sentences, in response to which the computer will generate syntactically correct Java source code. We believe that such a system would help beginning students by allowing them to focus on concepts instead of syntactic details, and that it would also be a boon to students with visual or mobility impairments. As a prelude to designing and implementing such a system, we evaluated the concept via a Wizard of Oz study. Volunteer subjects were told that they were helping us evaluate a working system. In reality, an accomplished programmer was playing the role of the purported system, and we were studying how the subjects interacted with it. We describe the system that we envision, discuss the process of running a Wizard of Oz study in the context of our own recently completed study, and summarize our preliminary results.*

Index Terms — Spoken English programming interface, Wizard of Oz study, natural language processing, teaching introductory programming.

INTRODUCTION

The scenario: You are a student in an introductory programming class. We tell you that we have developed a system that allows you to code by sitting at a computer monitor and describing a program into a microphone. You agree to help us evaluate the system. We take you into a room with a computer, which has neither a keyboard nor a mouse, and give you general instructions about using our system. We then ask you to use the system to complete your current homework assignment.

The problem: The system doesn't exist. In fact, we need to find out how students would prefer to interact with such a system before we can design and develop it.

What you don't see: Your computer is connected over a network to a second computer. An expert sits at the second computer and pretends to be the system. The expert hears (and sees a transcript of) everything you speak, determines what the system's response should be, and displays this response back on your computer.

This is the basic outline of a *Wizard of Oz study* that we spent a year and a half creating and running. A Wizard of Oz study allows researchers to evaluate the 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 [2]. 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.

We thought that our experience with the study would be of interest to the FIE community for two reasons. First, the long term goal that is served by the study—the creation of a system that would allow beginning students to program by speaking instead of typing—could impact the delivery of introductory programming classes. Second, we had no experience with Wizard of Oz studies and no idea what we were getting into. We have learned a lot during our study, and we would like to share our experiences.

NATURALJAVA

We are in the early stages of developing a spoken language interface for writing 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. We believe that this system will be very useful for students learning to program, as well as for programmers with vision and mobility impairments.

[1]David E. Price, University of Utah, School of Computing, deprice@cs.utah.edu

[2]Dana Dahlstrom, University of California at San Diego, Department of Computer Science and Engineering, dana@cs.ucsd.edu

[3]Ben Newton, University of Utah, School of Computing, bnnewton@cs.utah.edu

[4]Joseph L. Zachary, University of Utah, School of Computing, zachary@cs.utah.edu

The work described here was supported in part by the NSF under grants IIS-0090100, DUE-0088811, and IIS-9704240.

0-7804-7444-4/02/$17.00 © 2002 IEEE

We currently have a working prototype, called NaturalJava, that takes typed input from the keyboard [3]. NaturalJava works within a subset of Java and, more to the point, within the small subset of English that its developers use to describe Java programs. Part of a typical interaction with NaturalJava appears in Figure 1; the resulting source code appears in Figure 2.

To produce the next version of our system, we plan to use the output of a commercial speech recognition system to drive NaturalJava. Before investing the effort to design and implement this version of the system, we needed to find out how its intended users would want to write and edit source code using spoken English.

STUDY OVERVIEW

We created a simulation of a full spoken language interface for programming. Our goal was to record the interactions between our system and a group of beginning programmers. Now that the study has been completed, we are beginning to analyze those interactions to determine how students tend to describe program creation and what vocabulary and sentence structure they prefer. We will also find out how well a commercial speech recognition system performed at recognizing students' speech and learn how the students perceived the utility of the system. Finally, we will see if the subjects' approach to speaking to the system evolves with experience. The results of the analysis will drive the design of the final system.

We designed and conducted the study over a period of about 15 months. Between October 2000 and May 2001, we implemented, and tested the hardware and software infrastructure required for the study. During the summer semester of 2001, we tried out the system with a single subject from an introductory Java programming class. During the fall semester of 2001, eight subjects from an introductory C++ programming class used the system. (We chose students from a C++ class because no Java class was offered; fortunately, Java and C++ are syntactically 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. 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.

During the study we recorded approximately 200 hours of interactions between subjects and the system. We are transcribing each session for future analysis.

Below we sketch how our study looked from the points of view of both the subjects and the expert. See [1] for more details about the underlying infrastructure.

1. Create a public method called deq that returns a Comparable.
2. Declare an int called i and initialize it to 1.
3. Declare an int called minIndex and initialize it to 0.
4. Declare a Comparable called minValue and initialize it to elements' firstElement cast to a Comparable.
5. Create a loop that iterates while i is less than elements' size.
6. Declare a Comparable called c and initialize it to elements' elementAt applied to i cast to Comparable.
7. Create an if statement controlled by c's le applied to minValue.
8. Assign i to minIndex.
9. Assign c to minValue.
10. Leave this loop.
11. Invoke elements' removeElementAt with minIndex as a parameter.
12. Return minValue.

FIGURE 1
INTERACTION WITH NATURALJAVA PROTOTYPE.

```
public Comparable deq () {
  int i = 1;
  int minIndex = 0;
  Comparable minValue =
    (Comparable)elements.firstElement();
  while (i < elements.size()) {
    Comparable c =
      (Comparable)elements.elementAt(i);
    if (c.le(minValue)) {
      minIndex = i;
      minValue = c;
    }
  }
  elements.removeElementAt(minIndex);
  return minValue;
}
```

FIGURE 2
SOURCE RESULTING FROM INTERACTION IN FIGURE 1.

The Subjects' View

We provided the subjects with a simple one-window user interface, which was divided into three regions: the code region, the prompt region, and the 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 only input device available to the subjects was a headset with a boom microphone—we removed the keyboard and mouse. A typical interaction with the system went as follows:

- Initially, the prompt reads "Please state your next request."
- The student says "Give me a for loop that goes from zero to ten new paragraph." (The subjects were instructed to say "new paragraph" to mark the end of each request.)
- The prompt changes to "Processing, please wait" while the system determines a response.
- A message appears in the message region: "Name of index variable?"
- The subject says "i new paragraph".
- The program in the code region changes to contain `for (int i =0; i <= 10; i++) {}`
- The initial prompt is redisplayed.

The Wizard's View

The hidden expert, who we will call the wizard, wore headphones and sat at a computer in a different room. The wizard manipulated three windows: a text editor in which he composed the source code, a text editor in which he composed messages, and a read-only window that displayed the output of the speech recognizer.

The student'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 student said) and logged to a text file;
- recorded to an audio file.

Each time a command was issued, the wizard responded by either modifying the code or composing a message, as appropriate. When the wizard saved the response to a file, the subject's display was updated. All source code modifications and messages were logged.

ORGANIZING A STUDY

The amount of preparation that was required to organize the study surprised us. In this section we summarize the organizational effort.

Get Institutional Review Board Approval

A Wizard of Oz study uses human subjects, so you must get approval from your Institutional Review Board (IRB). You will be deceiving your subjects, which is acceptable to the IRB so long as you inform your subjects of the deception after the study is completed. Make sure you get approval to delay informing the subjects of the deception until the study is complete. If anyone finds out before the study is finished, word could spread to other subjects and invalidate future data.

Build an Interface

For the deception to be effective, the interface must be convincing. The interface cannot show any signs of the human behind the scenes. Screen updates must be instantaneous—the user shouldn't see the characters appearing one-by-one. Conferencing software is not a good solution because mouse and editor cursors will move in an obviously human way. This is why we decided to have the wizard commit all changes to a file, and to only then have the students' interface display these changes directly from the file.

More subtle indicators of human intervention must also be obscured. For example, we were careful not to let the students see the output of the speech recognition software. Subjects seeing poorly recognized speech leading to perfectly generated code would quickly become suspicious.

Select Good Wizards

Good wizards are crucial. They spend every session reacting to whatever the subject does. It is impossible to provide guidelines to the wizards telling them how to respond in every conceivable situation. The wizards will have to make unsupervised decisions very quickly and exercise good judgment in choosing an appropriate response. Furthermore, the wizards must be consistent: once they provide some functionality, they must continue to provide that functionality for the rest of the session. (Any loss of functionality between sessions can be explained as the result of correcting a problem found by another subject.)

Find a Good Con Man to Front the Study

There will be a lot of lying, so you need someone who can lie quickly and convincingly. Equally important, the lies told must be consistent. For instance, one day we had a network problem that allowed data to flow from the subject's machine to the wizard's machine, but not the other way. As we did our troubleshooting of the problem, we had to tell a sequence of consistent (but increasingly more elaborate) lies to explain our actions without revealing the true nature of the problem.

Practice, Practice, Practice

It is important for the wizards to do many practice sessions before starting to work with subjects. They need to feel comfortable with the interface, and every response needs to be automatic. The more comfortable they are with their various tasks, the easier things will be when the study actually starts.

Practice also provides a valuable opportunity to find and work out bugs with the subjects' interface. For example, our practice sessions showed that the subjects tended to focus on the source code window while waiting for a response from the system. Therefore, the subjects were missing messages being sent to them to request more information. Consequently, we decided to have the message window flash red when a message arrived. This avoided the problem of the subject sitting idly waiting in vain for a response in the source window, with the wizard powerless to do anything.

Get Your Story Straight

Think out the basic structure of the fictional system. When we first solicited volunteers, they asked how the system worked. We couldn't say, "Well, you see, there is this person at another computer pretending to be this system ..." It is important to have a plausible story, but we recommend giving as few details as possible. It is amazing what students will remember. Several students continued to press for details about aspects of our system for weeks after our initial explanation.

RUNNING THE STUDY

Running the study requires a great deal of time and energy. Much of the effort here centers on recruiting, training, retaining, and debriefing the subjects.

Recruiting Subjects

One of our biggest problems was recruiting subjects from introductory programming classes. Our goal each semester was to recruit ten subjects. During the summer of 2001, we offered the students the incentive that participating in the study would presumably make it easier for them to complete their assignments. Unfortunately, we ended up with only one volunteer.

There were several reasons why students were reluctant to volunteer. First, many of our undergraduate students work full-time jobs and try to limit their time on campus as much as possible. Spending two hours per week for a session in our study was not appealing. Second, many students put off working on an assignment as long as possible. They don't want to commit to an additional two hours in their schedule because they may need that time to work on some other assignment. Finally, some students probably fear getting involved in a research project because they aren't sure what it will entail.

During our testing phase, we had trouble getting subjects to show up regularly. This was frustrating, since the front man's and the wizard's time was wasted. This problem was in part a lack of adequate incentive, and in part a problem of procrastination. If there is no strong incentive to show up, and if some other distraction arises, the subject can easily choose to skip a session.

During the fall semester of 2001, we changed tactics and offered to pay the volunteers $20 per session for their participation. We were able to recruit eight subjects. With only one exception, we had no trouble getting the subjects to show up regularly. In fact, some students asked us to make up tasks even when they had no homework assignments.

Initiating Subjects

The purpose of a Wizard of Oz study is to find out how subjects would like to use the system being developed. Giving the subjects too much instruction or too many examples will bias how they use the system. The subjects will attempt to duplicate your instructions or examples instead of doing things their own way. Alternatively, giving the subjects too little information may result in the subjects doing uninteresting things.

Initially, we did not provide enough information to the subjects. This 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 us with interesting data.

After the first group of sessions, we settled on a formula that worked well. We gave the subjects a way to think about working with our program and one concrete example of how to use it. We 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.*

We 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, our subjects had enough information to start working with our system using a larger vocabulary and a wider variety of sentence structures.

Reaction of the Subjects

During our study, we received feedback from subjects in two forms. At the end of each session, we asked the subject to move to a computer with a keyboard and fill in a web-based evaluation form. We also received spontaneous verbal feedback from subjects during their sessions.

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. We received a number of useful (and amusing) replies. One student, for example, appreciated the help that the system gave with C++ syntax:

> *I also was impressed with how well it understood what to do with pointers, it understood better than I did. I just told it what I thought I wanted and the assignments were made correctly.*

Another student complained about the error feedback:

> *When it says "Command not understood" I'm left in the dark. Did I mumble, was it bad wording, was my command too long, was there a specific word that it wasn't able to understand, or what?*

A third student was a bit greedy:

> *Also it would be great if the program knew how to write functions. If I could just tell them what I wanted my variables to be, and the idea that I want each function to do, and it did it, that would be good.*

The subjects often forgot they were being recorded; their unguarded comments give us 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 subjects perceived problems.

Such comments, taken in conjunction with the related changes to the source code, will help us sort out what users meant when they employed ambiguous language. 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").

We also got evidence that some subjects were deliberately challenging our system. One woman declared ten variables in a single command, counted them, and then said under her breath, "Let's see if it can do this."

PRELIMINARY RESULTS

The subjects expressed great enthusiasm for the natural language programming interface. After finishing a session, one student asked, "When can I buy this software?" While responding to the post-session questionnaire, another student wrote, "The feature that I liked the best was the automatic syntax. It was very convenient to speak as if I was talking to another person."

After each session, we ran a script to collate the data we collected. These data included the recording of the subject's requests, the transcription by the speech recognition software, messages/questions that passed from the wizard to the subject, and the changes made to the source code file. The collation allows us to view a command-by-command evolution of the program, so that we can see each input and its results. We are currently manually transcribing everything that each subject said for later study. A complete set of transcripts is not yet available, and we are just beginning to analyze these data. However, we can already draw some preliminary conclusions.

Subject Evolution

The transcripts that we have completed to date reveal that, over time, the subjects became better at expressing themselves to the system. In early sessions, for example, some of the subjects tended to dictate syntax; in later sessions, these same subjects began to phrase program descriptions at a higher level. For example, subjects commonly needed to insert `#include <iostream>` into their programs. Here is how one subject issued the same request in his first and last sessions:

- *Pound include less than iostream greater than.*
- *Include iostream.*

After the first few sessions, the subjects gained the confidence to phrase requests in English rather than by dictating syntax. Even so, there are many examples of a subject's ability to precisely describe a language construct improving with time. The following requests, all of which involve creating loops, were extracted from the transcripts of a single subject. One or more weeks passed between each successive transcript.

- *Create a fuh fuh fuh a loop that increments x by one and which multiplies two integers x times y.*
- *Create a for loop. The three arguments are for [pause] for temp x equals x2 next argument temp x is less than x2 then the next argument is temp x incremented by one.*
- *On the next line create a for loop that initializes an int loop = zero tests the loop while it's less than five and then increments by one.*

The subjects probably spent more time modifying code they had already written than they did writing new code. Compare the following two excerpts from one subject's transcripts. The first, which is composed of five separate requests, is from a session early in the semester. The second is from a session late in the semester.

- *Go up two lines. Go to the right six spaces. Go to the right one space. Backspace one time. Insert capital v.*
- *Go up two lines and change the one to a negative one.*

On reflection, these results are not surprising. We have observed that most beginning programmers have a hard time talking about their programs, whether they are asking questions or describing a solution. To make the best use of a natural language programming system, it may be necessary to explicitly teach beginning students a basic vocabulary for talking about programs.

Speech Recognition Performance

A comparison of the automatically and manually generated transcripts shows that the speech recognition software performed poorly in our study. Here are three pairs of commands issued by one of our subjects. The first command in each pair is what the subject actually said; the second is the output of the speech recognition software.

- (Spoken) *Go up two lines and send g to the draw array method.*
  (Recognized) *Go to liens and extended GE to the draw our ray method.*
- (Spoken) *Create a private array of ints named array.*
  (Recognized) *Create a private array of Inns named our ray.*
- (Spoken) *Create a for loop that has an int loop equal zero as the first parameter second parameter loop is less than fifty third parameter loop plus plus.*
  (Recognized) *To create for loop that has an into loop equal zero as the first parameter second parameter loop is less than 53rd parameter loop plus plus.*

The poor performance of the speech recognition software is not surprising because we didn't exploit its full capabilities. A new user is supposed to train the system in two steps. First, the user reads canned text that comes with the system. Next, the user reads text that is representative of his or her work, then interactively corrects the errors. We skipped the second step of the training because it would have biased the subjects' choices of words during the study.

Since the recognized speech wasn't displayed to the subjects, they couldn't correct the recognition errors that were made by the speech recognition system or be self-trained to speak in a manner that improved recognition. To complicate matters, speech in the programming domain uses words (such as "int") and phrase constructions (such as "for loop") that do not exist in standard English.

Offline from the study, we have experimented with training the speech recognition system using text from the programming language domain. It is possible to tune the speech recognition system so that it is capable of recognizing speech despite the unusual sentence constructions used to describe programs.

Disfluencies and Restarts

Disfluencies (*e.g.*,w ords like "um" and "er") and restarts (starting over in mid-sentence) pose a significant challenge to natural language processing systems, even if the underlying speech recognition is perfect. Our transcripts reveal a number of disfluencies and restarts. Here are two examples:

- *Change the word void no skip that change the word public to private.*
- *Go up four lines and create a aa a an init method.*

We believe that we can deal effectively with the problems posed by disfluencies and restarts. First, we are using a shallow parser in the prototype that will not fail catastrophically with ungrammatical sentences. Second, information extraction techniques work well over sentence fragments, and if we lose occasional items due to disfluencies and restarts, we can determine what information is missing from a command and query the user for it. Finally, we can integrate the speech recognition system's correction and editing functions into the programming interface to help minimize the problems posed by disfluencies and restarts.

CONCLUSIONS AND FUTURE WORK

The subjects in our Wizard of Oz study reacted positively to their experiences. Their post-session comments included many critiques of the strengths and weaknesses of the simulated system. On balance, these comments have strengthened our conviction that a spoken language based programming interface would benefit beginning programming students.

We have transcribed most of the approximately 200 hours of audio recordings. A preliminary review of the transcripts shows that students tend initially to describe programs in purely syntactic terms, and to then slowly develop the vocabulary required to describe their programs at a more abstract level.

Our preliminary review also reveals that the untuned speech recognition software performed poorly during the study. Offline of the study, however, we have found that the speech recognizer can be tuned to work with the kind of speech that is used to describe programs.

We will soon finish transcribing all of the subjects' utterances. We will then analyze these transcripts, in conjunction with the source code the subjects generated or modified, to determine the vocabulary and sentence structure that the subjects employed and to learn how they described programs. We will then feed this information back into the design and implementation of a working spoken English programming system. Eventually, we hope to repeat this user study with a real system.

ACKNOWLEDGMENT

Ellen Riloff provided many insights and suggestions. Hung Huynh worked as one of the wizards. The study would have been impossible without the help of a number of volunteer undergraduate subjects.

REFERENCES

[1] Dahlstrom, D. A system for wizard of oz studies in natural language programming, Bachelor's Thesis, School of Computing, University of Utah (2001).

[2] Moore, R., and Morris, A. Experiences collecting genuine spoken enquiries using woz techniques. In

Fifth DARPA Workshop on Speech & Natural Language (1992).

[3] Price, D., Riloff, E., Zachary, J., and Harvey, B. NaturalJava: A natural language interface for programming. In *Proceedings of the Conference on Intelligent User Interfaces* (2000).