# An Assessment of a Speech-Based Programming Environment

Andrew Begel*
Microsoft Research
One Microsoft Way
Redmond, WA 98052
andrew.begel@microsoft.com

Susan L. Graham
University of California, Berkeley
776 Soda Hall #1776
Berkeley, CA 94720-1776
graham@cs.berkeley.edu

## Abstract

*Programmers who suffer from repetitive stress injuries find it difficult to program by typing. Speech interfaces can reduce the amount of typing, but existing programming-by-voice tools make it awkward for programmers to enter and edit program text. We used a human-centric approach to address these problems. We first studied how programmers verbalize code, and found that spoken programs contain lexical, syntactic and semantic ambiguities that do not appear in written programs. Using the results from this study, we designed Spoken Java, a syntactically similar, yet semantically identical variant of Java that is easier to speak. We built an Eclipse IDE plugin called SPEED (for SPEech EDitor) to support the combination of Spoken Java and an associated command language. In this paper, we report the results of the first study ever of any working programming-by-voice system. Our evaluation with expert Java developers showed that most developers had little trouble learning to use the system via spoken commands, but were reluctant to speak literal code out loud. As expected, programmers found programming by voice to be slower than typing.*

## 1. Introduction

Interaction with software development environments can be frustrating for the growing numbers of developers who suffer from repetitive strain injuries (RSI) and other disabilities that make typing difficult or impossible. Speech interfaces can be used to help developers reduce their dependence on typing, reducing the onset of RSI among computer users, and increasing access for those who already have motor disabilities. Speech-based programming also may provide insight into better forms of high-level interaction.

Our early work in this area studied programmers to find out how they might speak Java code in a programming situation [3]. We found significant differences between written and spoken code mainly in areas of lexical, syntactic, semantic, and prosodic ambiguities that appeared in speech. We used these results to design Spoken Java, a new input form that balances the programmer's desire to speak Java naturally with the ability of our system to understand it. Spoken Java looks very much like Java, but consists only of the natural language words in the Java language, making all punctuation optional. Due to its human-centric design, Spoken Java can appear quite ambiguous from the computer's point of view. We have addressed that problem by applying program analysis to both the code already written and the code being spoken [1, 2]. We use the additional contextual information provided by the code to filter out incorrect and inappropriate interpretations, leaving the human programmer to intervene only when the computer cannot fully disambiguate a particular utterance.

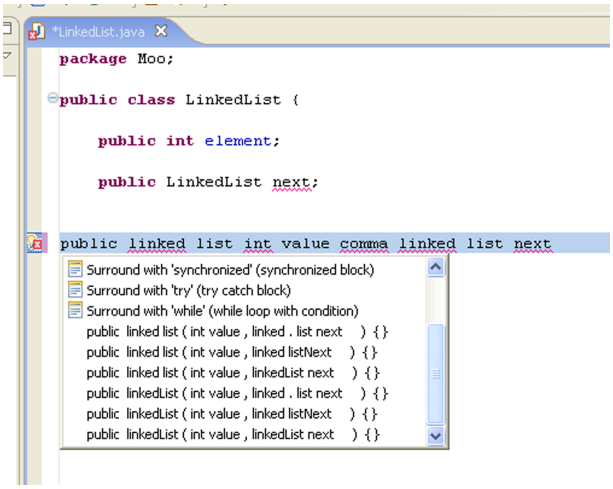


**Figure 1.** A screenshot of SPEED. The user has just spoken a constructor declaration for the LinkedList class. SPEED presents a pull-down menu for the user to choose the correct interpretation.

*This work was done while the first author was at the University of California, Berkeley.

In this paper, we report on a study to understand the cognitive effects of spoken programming in an editor we designed, called SPEED, or SPEech Editor [1]. SPEED is our speech-enabled Java program editor embedded in the Eclipse IDE [6] (see Figure 1). We asked several experienced professional Java programmers to use SPEED to create and edit a small Java program. We ran two versions of the same study, one with a commercial machine-based speech recognizer, and one with a human simulating the machine-based speech recognizer. We found that the programmers were able to learn very quickly to write and edit code using SPEED. We anticipated that programmers would often dictate literal code, but found that they preferred describing the code using code templates. The programmers were reluctant to speak code out loud. We expected that programmers would find speaking code to be slower than typing; this hypothesis was confirmed. Finally, programmers all felt they could use SPEED to program in their daily work if circumstances prevented them from using their hands.

The rest of the paper goes into more detail. In section 2, we present our usability study, its results, and implications for further design of programming-by-voice environments. We then survey the related work in this field and conclude.

## 2. Usability Study

We conducted a study to learn how expert Java programmers write and edit code using SPEED. The purpose of the study was to help us to understand how developers mix Spoken Java with commands, as well as to show us the kinds of commands the programmers use. In addition, we wanted to classify the mistakes that programmers and our system made that affect non-contiguous code entry and edits.

The study was conducted in two distinct sessions. The sessions were the same except for voice recognition technology. In the first, we used Dragon NaturallySpeaking. In the second, we used a non-programmer human to type in what the programmer said. The first session represents what can be done with state-of-the-art voice recognition tools with minimal training. The second session illustrates how SPEED could be used when the voice recognition accuracy is as close to perfect as it can get.

### 2.1. Participants and Task

The participants in the study were expert Java programmers familiar with Eclipse, with an average of twelve years of experience, drawn from the software development industry. Most of them had never used speech recognition software before; those who had used it abandoned it quickly because of poor accuracy. None of the participants had motor disabilities which would make typing difficult. The first session had three people; the second session had two.

We gave programmers 20 minutes to create a linked list class with an append method that takes two linked lists and merges them. Linked lists usually contain two fields, one pointing at the element, and another pointing at the rest of the list. The constructor builds a linked list node. In coding this simple data structure, any observed difficulties were likely to reflect the programmers' efforts to learn and use SPEED rather than their efforts to create the program itself.

### 2.2. Experimental Setup

SPEED was deployed on a Pentium 4 3.2 GHz computer with 2 GB of RAM. Programmers used a Plantronics DSP-300 microphone. Screen captures of the sessions were taken with Camtasia Studio 3, while the developers' voices were captured on a second computer.

The first session's software developers had a 15-minute Dragon NaturallySpeaking 8 voice training process and a 15-minute SPEED training session before they began their tasks. Developers were given a paper crib sheet with commonly used commands printed in a big font. The second session skipped the voice recognizer training process.

During the second session, a human volunteer was recruited to *be* a voice recognizer. He was set up behind the study participant, facing the opposite direction. His display was connected to the study computer by VNC [9], allowing both him and the study participant to see what was happening on the screen at the same time.

The human voice recognizer was trained for 45 minutes in order to practice interpreting Spoken Java commands, and to learn when to just type in what the user spoke. To assist in translation, the human was also given a crib sheet indicating the mapping between command phrases and keystrokes used to activate those commands. Punctuation and other non-words were entered in an arbitrary way by the human as he saw fit. Depending on the brand of voice recognizer, a software client may receive a punctuation mark or the spelled out punctuation. So, this setup is similar to the kind of text that the voice recognizers return.

### 2.3. Evaluation Metrics

The users' programming sessions were analyzed on a number of metrics shown in Table 1 and Figure 2. Metrics were coded by one viewer watching the screen capture while listening to the audio recording of participants performing the tasks. Each participant's record was played back three times to confirm the measurements.

### 2.4. Hypotheses

We hypothesized that SPEED users would follow a typical programming pattern: they would navigate through the

| | | U1 | U2 | U3 | U4 | U5 |
|---|---|---|---|---|---|---|
| | | Machine VR | | | Human VR | |
| **All Utterances** | | 131 | 125 | 275 | 102 | 105 |
| **Commands** | | 73 | 105 | 206 | 68 | 70 |
| **Dictated Phrases** | | 58 | 20 | 69 | 34 | 35 |
| **Correctly Recognized** | | 66 | 97 | 164 | 81 | 92 |
| **Recognition Error Rate** | | 50% | 22% | 40% | 21% | 12% |
| **VR Mistakes** | **Extra Words** | 6 | 0 | 3 | 0 | 0 |
| | **Wrong Words** | 8 | 3 | 34 | 1 | 3 |
| | **Did Not Hear** | 13 | 15 | 42 | 1 | 0 |
| **SPEED Mistakes** | **Bug** | 5 | 0 | 2 | 2 | 4 |
| | **Design Flaw** | 1 | 2 | 0 | 2 | 0 |
| | **Crash** | 4 | 6 | 5 | 2 | 4 |
| **User Mistakes** | **Did Not Know** | 2 | 0 | 2 | 5 | 2 |
| | **Wrong** | 2 | 2 | 3 | 0 | 3 |
| | **Ungrammatical** | 2 | 2 | 11 | 1 | 0 |

**Table 1.** Data recorded from SPEED User Study from all participants. The first section displays aggregate statistics, while the second classifies errors as they were made during the session.

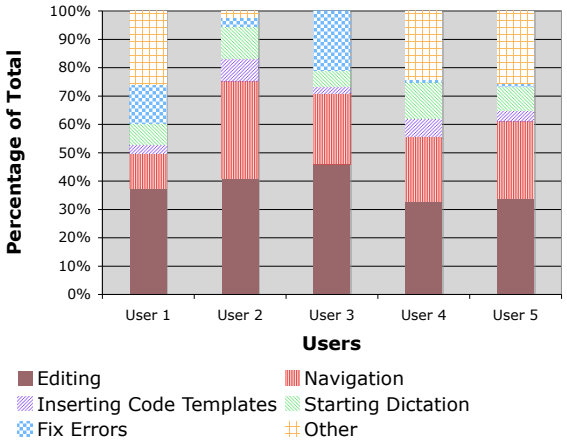


**Figure 2.** Distribution of Spoken Java commands spoken for various purposes.

document to a desired insertion point, activate Spoken Java, and add new code or edit existing code. A few of the editing commands would be used in the majority of situations, so we anticipated that learning the commands would take only a few repetitions. Based on our earlier user studies, we anticipated that speaking the Spoken Java language would be intuitive and natural for programmers without any training. Our GOMS analysis predicts that programmers should work more slowly when programming by voice than by keyboard due to the slow speed of speech recognition compared to typing. Finally, based on the results of Christian et. al. [4], we thought that programmers would sometimes forget what they were doing because of anticipated cognitive interference between speaking and thinking about code.

## 2.5. Results and Discussion

**Speed and Accuracy:** The programmers in the first session had a decidedly different experience than those in the second session. Stymied by the slow speed and poor accuracy of the voice recognition software, they were able to accomplish only a portion of the code creation task in the time allotted. They created a class, some fields and a constructor, but were not able to fill in the code for the constructor or create any methods. Using the human voice recognizer, the second group were able to complete far more of the task. They were able to complete two classes (a list node and a list class), each with several fields, a constructor to initialize the fields and the beginnings of an append method.

The accuracy of the machine voice recognizer after limited training was abysmal. At times, the recognizer would just not hear anything the programmers said. Other times, it would recognize a series of commands perfectly. Dictation was mostly unusable. Breathing was often interpreted as a single syllable word, so programmers learned to prevent that by muting the microphone. Often, words were recognized poorly, coming close to what the user wanted, but not close enough. For example, "linked" came out as "links." Most programmers said `Scratch That` to undo the utterance and tried again. The second group experienced almost error-free recognition, contributing significantly to a reduction in the number of commands uttered to fix mistakes.

Recognition delay was about the same for both groups, between 0.5 and 0.75 seconds. We imagine that while faster computers will reduce recognition processing time further, the human reaction time will remain the same. A major component of the software speed problem was Camtasia Studio. When it was recording, the software slowed down by at least three times. One participant who got to use SPEED for a short time without screen capture reported that the speed was perfectly fine.

Due to the speed and accuracy problems, participants in the first session adopted a stop-and-go pattern of speaking and waiting for the results. Their mistrust of the voice recognizer caused them to program very slowly and increased their frustration with SPEED. The second group were able to speak at a normal pace, and often paused in the middle of commands as they were uttered, something a machine speech recognizer would have never recognized correctly.

**Spoken Java Commands:** All users learned the SPEED commands fairly quickly, requiring only one or two repetitions of each command to use it without looking at their crib sheet. Command error rates caused by user error were fairly low for all participants. Users made an average of 6 mistakes where they did not know what to say or how to say it. The first group had recognition error rates between 22% and 50%, which made the system unusable. The second group enjoyed 12% to 21% errors. Since the errors made by these users were often their own, rather than caused by misrecognition, they were more acceptable to the users.

In the first session, programmers tended to speak in short bursts due to recognizer malfunction. When the wrong word appeared in dictation, programmers had to spend time to correct it. This effect was not seen in our spoken programs study [3] because participants were talking into a tape recorder, and could not see the results of the transcription.

Figure 2 gives a breakdown of the distribution of Spoken Java commands used for different purposes. The majority of commands spoken were for editing and code template insertion. Editing commands were mainly used in place of string editing facilities that are usually performed by repetitive single keypresses, such as deleting a character from a word, inserting spaces between words, or changing the capitalization of a word. As predicted by a GOMS analysis [1], spoken commands that cause only a few letters to change on the screen cannot compete with typing for efficiency. Code template insertion was thus seen as providing a lot of text for very few words. The small percentage of code template commands that were seen achieved the largest payoff for the programmer, and are almost directly correlated with the number of program structures created by the study participants. Code dictation was almost always used to edit the name of a field, method or type name. Filling in code templates was a simple process that involved navigating to the next slot, saying `Edit This`, dictating the new name, and then saying `Done`. Since most new names were single words, SPEED was able to automatically translate what users said from Spoken Java to Java without any need for interaction. Multi-word identifiers were concatenated automatically, using our speech-aware programming language analyses, enabling developers to skip tedious editing commands for spacing and capitalization. Our system does not perform capitalization of the first letter of a word automatically (encoding a style convention in Java that class names start with a capital letter), requiring users to speak `Cap That` on many identifiers.

Participants had suggestions for better commands. A `Jump To` command could let the user speak code from the screen and have the SPEED cursor jump there. This is very similar to Select and Say from Dragon NaturallySpeaking. Several wished that the code templates could take a parameter with the name of the item being created, for example, `insert field element` instead of `insert field`. Our code templates only worked on blank lines. One user wanted the commands to insert templates in a stylistically appropriate location (for example, putting fields at the top of a class) even if spoken in the wrong context.

A few asked to customize the command names used to insert code templates. However, one participant explicitly said there should be no customization in order to make it easier for all users of a speech programming tool to learn the same language, as well as to make it easy to move from one speech-based programming environment to the next.

**Speaking Code:** Participants were apprehensive about speaking the natural language words in the program when dictating code, but not when saying identifier names. In fact, most dictation utterances were for identifiers. One called the idea of dictating code "strange." Four preferred to describe the code instead of dictating it. This is unexpected, but fascinating. One felt that describing the code was a higher level form of programming, apparently concluding that if higher-level programming was a good idea, then describing code must be as well. This result concurs with our own GOMS analysis [1], as well as with Snell's findings that code template insertion is a useful feature for programmers to most easily enter large amounts of code [11].

Two participants felt that tandem use of keyboard and voice would potentially be more efficient than either alone. Without specialized navigation commands, voice is inefficient at moving the cursor to a particular location on screen, especially inside pure text regions, but a keyboard and mouse make this simple. For code entry, voice could be more economical through code templates. Code templates are available by keyboard in Eclipse, but participants had trouble remembering the proper keywords to activate them. They thought the voice commands were easier to remember.

One user wanted integration of voice with Eclipse's code completion feature. When the user enters a method parameter's type, a list of all types could be shown in a popup menu. As the user speaks the words composing the type name, the menu would be filtered to show only matches. Once only one remained, the user could select it.

**Subjective Evaluation:** When asked whether they would consider using voice recognition for programming, most were apprehensive, pointing out problems with noise pollution, cognitive interference (speaking interferes with thinking), and wasting the use of their hands while speaking. One would only use voice recognition if everyone else were to do so. Another might use it at home if he worked alone, where he would not bother anyone else.

The participants concluded that none of them would use this programming environment for daily coding, especially with the poor accuracy provided by the machine voice recognizer. However, they all would consider using the software if they got RSI, worked from home, or were in a hands-free environment, such as while pacing around the room, or sitting with the keyboard unavailable. In spite of their reluctance to use this software, all programmers noted that since coding was not the primary part of their daily work, using a voice-based programming environment would not have a significant effect on their efficiency as a programmer.

## 3. Future Work

The user study showed that machine-based voice recognition performance after 15 minutes of training is inade-

quate. Programmers would have to train for many hours before recognition accuracy would improve enough to be usable. Obviously, continuing to use a human voice recognizer is not a viable solution. In an attempt to improve the accuracy, we retrained the recognizer on a Spoken Java corpus. Unfortunately, this introduced spelling errors in recognized words and did not improve recognition of commands.

Our study looked primarily at writing new code. Editing commands were used only to fix mistakes and edit identifier names. Large-scale code maintenance, and even small code motion operations, need to be speech-enabled and evaluated as well. Operations that invoke IDE commands are often speech-enabled (all voice recognizers can speech-enable menu items and GUI buttons), but the more common operations should be evaluated through a user study.

Participants' reluctance to use code dictation services needs further exploration. They claimed that they preferred higher-level coding actions such as template instantiation. Our spoken programs study [3] also showed this preference. Programmers tended to abstract the code on the paper, especially when there were obvious patterns. There were no obvious patterns to speak in the SPEED study, but programmers still wanted to use abstractions. However, code dictation would still seem necessary for code constructs that do not have programmer-understood names and for complex program constructs that programmers will only use once.

## 4. Related Work

Efforts to apply automatic speech recognition for programming tasks using conventional natural language processing tools have had limited success. Inside a text editor, vendor-supplied editing commands are oriented towards word processing, supporting style changes and clipboard access. Jeff Gray speech-enabled the Eclipse programming environment [10], but not the editor. T.V. Raman speech-enabled Emacs, making accessible all of the Meta-X commands and E-Lisp functions [8]. Raman also enabled Emacs to render its text and commands in spoken form.

Speech-enabling IDEs is only one step to making a usable programming environment. To author, edit and navigate through code by voice, developers need to speak fragments of program text mixed with navigation, editing, and transformation commands. VoiceCode is a notable success [5], using finite-state command grammars to provide support for Python and C++. VoiceCode has not yet been formally evaluated. NaturalJava [7] accepts natural language descriptions of programs, where programmers describe the Java constructs they wish to create instead of saying the code directly. NaturalJava only supports code authoring, not editing or navigation. NaturalJava's design was evaluated through a Wizard of Oz study. Our study of SPEED is the first and only study of voice-based programming in a working programming-by-voice environment.

## 5. Conclusion

We created a program editor called SPEED that supports programming by voice for code authoring, editing and navigation, and evaluated it through a user study. We found that programmers are able to learn to program verbally with little practice, but have significant trouble when the speech recognizer misinterprets what they say. Programmers prefer high-level abstraction to code dictation, and perceive speech-based programming to be less efficient than typing, but efficient enough to perform their daily work.

Programming-by-voice can enable motor-impaired software engineers to program, albeit at reduced efficiency compared to an unimpaired programmer. With more study, different user interface designs, and better analysis, software developers will one day be able to use speech-based programming to compete effectively in the workforce.

## References


[1] A. Begel. *Spoken Language Support for Software Development*. Ph.D. Dissertation, University of California, Berkeley, 2005. Report UCB-EECS-2006-8.

[2] A. Begel and S. L. Graham. Language analysis and tools for ambiguous input streams. In *LDTA*, 2004.

[3] A. Begel and S. L. Graham. Spoken programs. In *VL/HCC*, September 2005.

[4] K. Christian, B. Kules, B. Shneiderman, and A. M. Youssef. A comparison of voice controlled and mouse controlled web browsing. In *ASSETS*, pages 72–79, 2000.

[5] A. Desilets, D. C. Fox, and S. Norton. Voicecode: An innovative speech interface for programming-by-voice. In *Proceedings of ACM CHI 06 Conference on Human Factors in Computing Systems*, April 2006.

[6] Eclipse. http://www.eclipse.org.

[7] D. Price, E. Rillof, J. Zachary, and B. Harvey. NaturalJava: A natural language interface for programming in Java. In *IUI*, Short Paper/Poster/Demonstration, pages 207–211, 2000.

[8] T. V. Raman. Emacspeak – direct speech access. In *ASSETS*, pages 32–36, 1996.

[9] RealVNC. VNC: Virtual Network Computing. http://www.realvnc.com/.

[10] S. Shaik, R. Corvin, R. Sudarsan, F. Javed, Q. Ijaz, S. Roychoudhury, J. Gray, and B. R. Bryant. SpeechClipse: an Eclipse speech plug-in. In *Eclipse Technology eXchange Workshop*, pages 84–88. ACM Press, 2003.

[11] L. Snell. An investigation into programming by voice and development of a toolkit for writing voice-controlled applications. M.Eng. Report, Imperial College of Science, Technology and Medicine, London, June 2000.