Programming By Voice: A Domain-specific Application of Speech Recognition

Andrew Begel
University of California, Berkeley
573 Soda Hall #1776
Berkeley, CA 94720-1776
(510) 642-4611
abegel@cs.berkeley.edu

Introduction

Programming environments can create frustrating barriers for the growing numbers of software developers that suffer from repetitive strain injuries (RSI) and related disabilities that make typing difficult or impossible. Not only is the software development process comprised of fairly text-intensive activities like program composition, editing and navigation, but the tools used for programming are also operated textually. This results in a work environment for programmers in which long hours of RSI-exacerbating typing are unavoidable.

Why is software development so text-oriented? Historically, programming languages have been typewritten, and the tools used to manipulate code were developed on text-only computer systems. While alternative non-textual languages have been developed over the years, none of these has had the huge commercial success enjoyed by more traditional text-based languages like C, C++ and Java. The primary tool used for programming is a specialized text editor. Early text editors were manipulated entirely via keyboard, and leveraged many keyboard shortcuts for common operations in order to speed the programming process. Unfortunately, keyboard shortcuts are usually composed of an alphanumeric key combined with one or more modifier keys (e.g. Control, Option, Alt), which contributes to RSI when keyed unergonomically. These days, text editors are usually embedded in integrated development environments (IDEs) that provide the programmer with integrated services, such as compilation and debugging. These program editors put more commands in menus and dialog boxes, increasing the interface's mouseability, but programmers quickly learn the keyboard shortcuts and go back to typing in order to increase their efficiency. All of these factors contribute to a poor work environment for programmers with RSI.

One way to reduce the amount of typing while programming is to use speech recognition. Speech interfaces may help to reduce the onset of RSI among computer programmers, and at the same time, increase access for those already suffering motor impairments. Many disabled programmers are already bootstrapping voice recognition into existing programming environments. However, speech does not map well onto the available applications and programming tasks. Our research uses a principled approach from field of programming languages to allow developers to use speech with much more powerful control. By exploiting the domain-specific nature of programming and applying programming language-based analyses to a verbalized form of authoring, editing and navigating through code, we hope to alleviate many of the common problems of voice recognition for programmers.

We also have other motives for the use of voice. People express things differently when they speak than when they write – by developing speech capabilities that are both comfortable for users and within our ability to process, we gain additional understanding of the vernacular suitable for high-level interaction with software artifacts. In addition, what we learn about speech applied to software development will inform us about the processing of voice input in other specialized domains. Finally, a voice interface may prove useful for presentation to groups (e.g. in the classroom) or for collaborative technologies such as pair programming.

Speaking Programs

Mainstream voice recognition tools such as IBM's ViaVoice and Scansoft's Naturally Speaking provide two kinds of services: command and control grammars and dictation. Both of

these services are poorly suited for understanding spoken program code. In command mode, an application uses a rule-based grammar to perform actions when the user says certain phrases. The command grammars are based on finite automata, not context-free grammars, and are not powerful enough to describe programming language syntax. Thus, command-mode solutions such as VoiceGrip [Desilets 01] suffer from awkward, over-stylized code entry, and the inability to exploit the structure and semantic meaning of the program. Keyword-triggered code template expansion and context-sensitive detection when the user is uttering an identifier has been shown to ease this awkwardness [Snell 00].

Speech-enabling text editing commands, as has been done by IBM, Scansoft and by contributors to public domain software [Raman 96], is not enough to support software development tasks. To perform these activities by voice, developers need to speak fragments of program text interspersed with navigation, editing, and transformation commands. Unfortunately, the commercial recognizers are based on statistical models of the English language; when they hear code, they turn it into the closest approximation to English that they can. This almost always results in misrecognition. The NaturalJava system uses a specially-developed natural language input component and information extraction techniques to recognize Java constructs and commands [Price 00, 02]. Portions of that work are promising, although at present, there are restrictions on the form of the input, and the decision tree/case frame mechanism used to determine system actions is somewhat ad hoc. Arnold, Mark and Goldthwaite propose to build a programming-by-voice system based on syntax-directed editing, but there are limitations to their technique as well [Arnold 00].

Our approach is to design the language around a natural verbalization of the code as it appears on the screen. Our main concern is to balance the ease of use of the language with the ability of our algorithms to understand it. A well-designed verbalization will need to describe both the program code and the commands needed by the programmer to manipulate and move through it. It also needs to be easy to learn, preferably using spoken constructs that the programmer already knows. But, would all programmers naturally speak the same language when they verbalize their program?

To answer this question, we conducted an experiment in which participants read a one-page pre-existing Java program out loud. We found that there does exist a common vernacular among programmers for speaking programs despite the diversity of their educational training. This enables us to create a verbalized programming language definition which will work for most programmers. However, some aspects of speech present challenges for system understanding. Punctuation, which is found in almost every programming language construct, is inconsistently verbalized, and is often omitted. Difficulties arise with homophones (words that sound alike but are spelled differently), capitalization of words, and concatenated words. When program text is spoken, words are dropped, repeated, misused or uttered out of order. We saw differences between native and non-native English speakers in regards to ambiguous utterances – native speakers use prosody (pitch and pausing) to disambiguate the construct, while non-native English speakers rephrase the construct in other ways. Native English speakers are also better at verbalizing abbreviations and partial words. We observed that programmers tend to identify patterns and describe them, rather than using only their instantiations. Other studies we have conducted show that navigation commands supplied by commercial voice recognition tools

suffer from several flaws: users must speak too many words, make repetitive utterances, and rely on generally poor human visual estimation skills [Begel 02].

Based on these experiments and studies, we have developed Spoken Java, a dialect of Java that is more naturally verbalized by human developers, along with a command and control language designed to enable programmers to find and select pieces of code and modify them in high-level linguistic ways.

Spoken Java

Spoken Java is designed to be semantically isomorphic to Java – despite the different input method a programmer uses to enter program text, the result is indistinguishable from a traditionally coded Java program. To be easy to use, most punctuation has been made optional, and all of the punctuation has English equivalents. For example, the open brace after a class declaration may be verbalized as "open brace," or more in line with what programmers actually say, "begin class." We have reversed the noun-verb ordering of the cast operator to be more compliant with the typical English syntax, "cast foobar to integer" rather than the more literal transliteration: "open paren integer close paren foobar."

Spoken programs are full of ambiguities caused by unverbalized punctuation. For example, when users want to extract an element of an array at a certain index, they write "a[i]". Most of the time, however, when users speak the same construct, the right bracket is not verbalized (e.g. "a sub i"), leading to an ambiguity. When the user wants to side-effect the value of the array element, they write "a[i]++" or "a[i++]" (the first increments the value of the array cell at index i, the second increments i). Users spoke both of these constructs identically as "a sub i plus plus," as though any listener could discern what they meant. This problem of unterminated expressions (where the right terminator is unspoken) exists throughout Spoken Java. Guided by our experiences with non-native English speakers, we have provided equivalent English-like alternatives (e.g. "element i of array a") to avoid using the ambiguous construct.

This ambiguity is just one of the many kinds that we have seen. Another important one arises from homophones, words that sound alike but are spelled differently. Here is an example: a programmer wishes to enter a loop construct to count from one to ten. In Java, the result should come out like this:

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

Typically, a programmer would verbalize the preceding code fragment without any parentheses, braces or semicolons, and thus say just the following words:

```
for int i equals zero i less than ten i plus plus
```

Unfortunately, the voice recognition tools provided by major commercial products, such as IBM's ViaVoice and Scansoft's Naturally Speaking, are poorly suited for speaking program code. These natural language speech recognizers are based on statistical models of the English

language; when they hear code, they turn it into the closest approximation to English that they can.

```
4 int eye equals 0 aye less then ten i plus plus
```

The word “for” is a homophone for the number “4” (or “fore” or “four”). The loop variable name “i” can be spelled “eye” (or “aye”), and the comparator “less than” may be misspelled “less then.” Unlike a human listener who can understand the intent of speech that contains mistakes, a program compiler cannot compile code containing any mistakes – the slightest error, for example, a misplaced character or misspelled name, can render the entire program invalid.

Understanding Spoken Programs

The mathematical precision of programming languages is both a curse and a blessing. It is a curse for verbal entry of programs because humans do not speak punctuation or capitalization, they drop and reorder words, and speak in homophones – all features of a program that must be precisely written down. All is not lost, however. The same precision that appears to hinder system understanding of spoken programs is also the solution. We can exploit knowledge of the program being written to disambiguate what the user spoke and deduce the correct interpretation. This cannot be done with natural language because natural language syntactic and semantic analysis is still infeasible, and natural language semantics are far more ambiguous than those of any programming language.

Using program analysis techniques we have adapted for speech, we use the program context to help choose from among many possible interpretations for a sequence of words uttered by the user. We present an example. A programmer wants to insert text at the cursor position in the following block of code:

```
String filetoload = null;
InputStream stream = getStream();
try {
    ▌
} catch (IOException e) {
    e.printStackTrace();
}
```

He says:

```
file to load equals stream dot read string
```

Let us look at the interpretation of just the first three words “file to load,” considering variable spelling and word concatenization. We can spell “to” as “too,” “two” or “2”. We can spell “load” as “lode.” And we can concatenate either the first two words together to make “fileto,” the second two words to make “toload,” or all three words together to make “filetoload.” This makes 32 possible interpretations of the words (8 spelling combinations times 4 word concatenizations) which are passed to our program analysis system.

The system next reads the input stream of words and computes all of the possible structures, taking into consideration the missing punctuation. Here are just sixteen possible structures for four of the possible input word sequences:

file to load	file 2 load	file tolode	filetoload
1. file to load	9. file 2 load	12. file tolode	15. filetoload()
2. file(to, load)	10. file(2, load)	13. file(tolode)	16. filetoload
3. file(to.load)	11. (file, 2, load)	14. file.tolode	
4. file(to(load))			
5. file.to(load)			
6. file.to load			
7. file to.load			
8. file.to.load			

The next phase of analysis uses semantic information about Java and the program to filter out invalid structures. Using the system's knowledge of the programming language, we can immediately rule out interpretations 1, 6, 7, 9, and 12 because in Java, two names are not allowed to be separated by a space. Next, after having analyzed the context around the cursor position, it can be determined what variable and method names are currently in scope, (i.e. visible to the line of code that the programmer is entering). If a name is not visible, it must be illegal, and therefore an incorrect interpretation. In our programmer's situation, there are no variables named "file," so interpretations 5, 8, 11 and 14 can be ruled out. Likewise, there are no methods (i.e. functions) named "file," so interpretations 2, 3, 4, 10 and 13 are incorrect. Finally, program analysis informs the system that there is no method named "filetoload," thus ruling out interpretation 15. The correct interpretation is 16, a variable "filetoload" where all three uttered words are concatenated together, and where the middle homophone is spelled "to," and the final homophone is spelled "load."

SPED: Speech Editor

Our system takes the form of a program editor called SPED (for Speech Editor) and associated program analysis framework called Harmonia [Boshernitsan 01] which are both embedded in the Eclipse IDE. A user begins by speaking some program code in Spoken Java into the editor. Once it has been processed by the voice recognizer, it is analyzed by Harmonia. Harmonia can recognize, handle and support ambiguities through the syntactic phases of program analysis [Begel 04], as well as execute semantic analyses to disambiguate the myriad possible interpretations of the input that the first two phases create. When semantic analysis results in several legal options, our programming environment defers to the programmer to choose the appropriate interpretation. Once the interpretations have been deduced, they are translated back into Java, and written into the editor.

In addition to composing and editing code, the programmer may perform high-level program manipulations. For example, he may invoke a program refactoring (a restructuring of the code)

or search for a particular structural or semantic entity in the code base by saying “Find all references to the MyList dot getElement method and replace them with StandardList dot get”. To support this combination of commands and code, we developed Blender, a lexer and parser generator that can merge descriptions of formally specified languages (for instance, from Spoken Java and its associated command language) into a tool that seamlessly recognizes the combination [Begel 04].

While we have striven to make Spoken Java as naturally verbalized as possible, there may be situations where the programmer does not know how to express a particular construct. Even complete knowledge of language syntax does not suffice to know the vocabulary, since the user will be able to create and select transformations and other higher-level mechanisms. In this case, we are developing a spoken feedback system where any already written construct may be spoken out loud to help teach and reinforce proper input techniques. We are combining spoken feedback with visual reinforcement of the language used by programmers as they enter each construct into the computer. These feedback mechanisms will help alleviate any short-term memory problems programmers may suffer when relying exclusively on voice input.

To mitigate problems in searching for code fragments, we introduce a context-sensitive mouse grid – a program-aware form of direct navigation. It allows programmers to “drill down” hierarchically through their program to select the desired statement or word without having to re-speak potentially difficult-to-verbalize program text. When verbalization is required, for example, when searching for a name in the code, we are developing a phonetics-based search to make it unnecessary for the user to spell the word precisely. Search results are all presented together, sorted numerically, and shown with surrounding context to enable the user to quickly navigate with a minimum number of utterances.

Evaluation will take the form of user studies of programmers composing and editing code. We will iterate the design of the interface, command language, Spoken Java, and the development environment's capabilities in response to these studies. We consider our work to be beneficial if our metrics show that users can perform better on our system than the current state-of-the-art.

Conclusion

Programmers suffering from RSI will always have a difficult time adapting to software development environments that promote long hours in front of a keyboard. Our work helps make this easier by enabling programmers to use voice recognition for all three programming tasks: composition, navigation, and editing. We have developed a means for exploiting the structure and semantics of the programming domain to create a naturally verbalizable form of Java that is amenable to analysis by voice recognition and compiler tools. We have built these analyses and enhanced our analysis tools to comprehend mixed command and programming languages. Our user studies will help inform the design of Spoken Java and help improve its command language. By applying specialized knowledge of the programming domain to voice recognition, we can create a program editor that will enable typing-impaired software developers to remain competitive in the workforce.

References

[Arnold 00] Arnold, Stephen, Mark, Leo and Goldthwaite, John. Programming By Voice, Vocal Programming. *In the Proceedings of the ACM 2000 Conference on Assistive Technologies*. November 2000.

[Begel 02] Begel, Andrew and Kariv, Zafrir. *SpeedNav: Document Navigation by Voice*. CS294-4 Assistive Technology Class Project Report. Spring 2002

[Begel 04] Begel, Andrew and Graham, Susan L. Language Analysis and Tools for Ambiguous Input Streams. *In the Fourth Workshop on Language Descriptions, Tools and Applications*. Barcelona, Spain, April 2004.

[Boshernitsan 01] Boshernitsan, M. *Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools*. M.S. Report. EECS, Computer Science Division, University of California, Berkeley. UCB/CSD-01-1149, 2001.

[Desilets 01] Desilets, Alain. VoiceGrip: A Tool for Programming by Voice. *International Journal of Speech Technology*, 4(2): 103-116. June 2001.

[Price 00] Price, David, Riloff, Ellen, Zachary, Joseph and Harvey, Brandon. NaturalJava: A Natural Language Interface for Programming in Java. *In the Proceedings of the International Conference on Intelligent User Interfaces*, January 2000.

[Price 02] Price, David, Dahlstrom, Dana, Newton, Ben and Zachary, Joseph. Off to See the Wizard: Using a "Wizard of Oz" Study to Learn How to Design a Spoken Language Interface for Programming. *In the Proceedings of the Frontiers in Education Conference*, November 2002.

[Raman 96] Raman, T. V. Emacspeak - Direct Speech Access. *In the Proceedings of the Second Annual ACM Conference on Assistive Technologies*, 1996, p. 32-36.

[Snell 00] Snell, Lindsey. *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.

Programming By Voice: A Domain-specific Application of Speech Recognition

Abstract

Programmers who suffer from repetitive stress injuries have difficulty staying productive in a work environment that requires long hours of typing. Programming By Voice helps to lower these barriers by enabling developers to reduce their dependence on typing by using speech. We exploit the domain-specific nature of programming, and apply programming language-based analyses to a naturally verbalized form of authoring, editing and navigating through code. We have developed Spoken Java, a dialect of Java that is more naturally verbalized by human developers, along with a command and control language designed to enable programmers to find pieces of code and modify them in high-level linguistic ways.