Program Commenting by Voice

Andrew Begel
University of California, Berkeley
abegel@cs.berkeley.edu

May 28, 2002

Abstract

Writing documentation is a perpetual exercise for the creators of software artifacts. For end-users, documentation is a key resource to learn how to use the artifact, but for the developer, documentation enables much more – not only the ability to understand someone else's code, but to document one's own thought processes regarding the architecture of the artifact and the code itself. Unfortunately, programmers usually do not create enough documentation, nor high enough quality documentation to replace an in-person discussion of the code. We believe this to be caused not by the inherent "laziness" of programmers to document their code, but by physical interference in the commenting activity, since both programming and commenting utilize the same input channel: the keyboard. We have created Commenting by Voice, a tool that enables programmers to create code comments using audio recording and speech recognition and have these audio comments inserted in their code with the same status as textual comments. We hope that by parallelizing the input channels, programmers will comment their code more, and in doing so, enable others to better understand the original thought processes involved in the coding task. Programmers are quite expressive writing code; if given a chance, we hope that they become as expressive talking about it.

1 Introduction

Documentation about software artifacts is at least as ubiquitous as the program code required to implement them. Programmers need many forms of documentation to do their work. For instance, programmers use API documentation to understand how to use libraries created on-site and purchased from outside vendors. Application designers create system architecture documentation to describe how all the components of a system fit together. Developers create engineering specifications to refine the client specifications in order to explicitly state the myriad implicit details that were left out.

In the source code itself, programmers create header documentation to describe the purpose of a file. Comments on code are also ubiquitous, and can span the gamut from a high-level description of an algorithm to low-level explanations for non-intuitive (read: clever) code. In this paper, we will concentrate on code documentation, in the form of program comments. We will justify this focus later in this section.

If documentation is so pervasive, why then is it almost universally perceived as being of such low quality? Communication and writing skills have often been found lacking in new college graduates in computer science. Since programmers have such bad writing skills, McArthur claims that they shouldn't be the ones writing any documentation for end-users of a system [13].

End-users are not the only consumers of documentation, as we saw above. Programmers themselves use documentation for many purposes, so we should ask, do programmers write good documentation for their own kind, and if so, what benefits are derived from the activity and the product? Detienne's studies [3] indicate that programmers who write comments before they begin coding seem to perform better on code comprehension tasks. Comments appear to aid in chunking, the process of grouping pieces of knowledge together. Unfortunately, there is a pitfall to commenting that can affect comprehension: comment management. When the comments don't appear in the code, but in a separate

document, one runs the risk, especially in larger software projects, of the documentation getting out of sync with the code that it is describing [11].

Sometimes there is too much documentation that focuses on details that experts find useless. Often this sort of documentation is written by novices because they themselves don't understand how a program works, so when told to comment their code, they concentrate on the lexical and syntactic pieces that they do understand [2]. Why are these kinds of comments (about a program's lexical and syntactic properties) less useful than comments about the semantics? Riecken et. al. performed a study on expert programmer's intentions for their comments [14]. First, they found that experts communicated semantic rather than syntactic knowledge about their program in the comments. This was because syntactic knowledge was assumed to be already understood by any reader (even complicated syntax was perceived to be a rite of passage for novices to understand), whereas the semantic knowledge was the hard part to understand and therefore the most critical to convey.

The flipside of too much documentation is too little. The fact that programmers don't write documentation until after they're done coding or perhaps never at all is well-known in the tech industry and is perhaps, one instigating factor of the many new programming methodologies that pop up every few years to encourage programmers to comment more [6].

1.1 Solutions

We can solve some of these documentation problems rather easily. If programmers are bad writers, we can just hire technical writers to write the end-user documentation for them. Better programmer education as well, is advocated to improve not only programmers' communications skills, but to improve their programming methodology and habits [19].

We can solve the out of sync documentation problem by inlining structured documentation in the code and process it with a separate tool to generate the final documentation. Such a technique is used by JavaDoc [10] for the Java programming language.

Some feel that tool support would help programmers document their code more easily [1]. There is a long history in automatic commenting tools [4, 17] which derive a description of the code through program analysis. More recently, an interactive commenting tool for Prolog was developed [15] that enables the programmer to comment on each step of execution of a Prolog query and insert those comemnts back into the code. Work with commenting agents has also been reported [5] which helps users design user interfaces.

1.2 Programmers are just lazy

However comprehensive this research seems, there is still one zebra left in our herd of horses. Programmers are perceived to be *lazy*. The unspoken argument is that they don't document because they don't want to expend the effort. We argue that this strawman argument is correct, but for the wrong reasons. Programmers are *not* lazy. In fact, programmers are anything but lazy. Who else would commit to the same long hours in pursuit of a bug or finishing off a feature before a deadline?

1.3 Or are they?

It is the assertion of this author that programmers comment poorly because there is no good time to do it. Why not? There are three times during coding when a programmer could comment her code: before she starts, while she is coding, and after she is finished. The kind of comments that can be written before the code are only blackbox comments – they must necessarily be descriptive of the intent and semantics of the code, for the lexical and syntactic structures haven't yet been written. After the programmer has spent a few hours poring over their solution, and the code is completely written, programmers feel as if they know the code like the back of their hands. It may even seem that there is no reason to comment code that appears so intuitively obvious (however non-obvious the code will appear in a week). Of course, the explanation is that "the source code *is* the documentation."

If the programmer can not be fully descriptive with their comments before they start, and does not want to or forgot to comment their code after she finishes, that leaves only one time to comment: while she is coding. We argue that programmers do not comment while coding as often as they should because coding and commenting use the *same input channel*: the keyboard. Thus, in order to comment their code, they must

necessarily stop coding, and vice versa. Even if they would like to be very descriptive with their comments about the actual code, in the end, it is the code that they get paid to write, not the comments.

1.4 Voice Comments

In this project, we aim to solve this input channel conflict. We enable the programmer to construct *voice comments* as they program by recording what the programmer says out loud into a headset microphone. We use a speech recognizer (IBM ViaVoice [9]) to translate the audio into text, and insert this audio/text combination into the code as a comment. Voice comments can be played back aurally or read visually at any time. The comments are saved (structurally) with the program document and restored when the document is reloaded.

By using the voice channel in addition to the keyboard channel, a programmer can talk about their code at the same time as they code it by hand. It is similar to a Think Aloud study, in which participants are encouraged to talk about what they are doing while performing an action. This enables an experimenter to gain insight into the thought processes involved in a task without cognitively interfering with the task itself. We hope to show that utilizing both voice and keyboard input channels will enable a programmer to annotate their code with spoken utterances about the thought processes that are going on in their heads as the they design and write down the program. In the rest of this paper, we present the design and implementation of the voice commenting project, as well as possible scenarios for its use in the real world. We then discuss experiments we would like to perform with novice and expert programmers to both better design the system and elucidate its impact on the programming process. Finally, we describe future work and conclude.

2 Scenarios

In this section, we present some possible scenarios of interaction with the Voice Commenting tool that we are building. The first is drawn from educational perspective, and the second from a code review perspective.

The notation that we use for visualizing the code is as follows. Program code is marked in `Courier font`. Traditional program commands are marked in *italics* and bordered by language-defined boundary tokens (e.g. in Java, we use */* A comment *\/*). Voice comments are marked in *italics* with ≪ and ≫ boundary tokens.

2.1 Education

It has long been the case that everyone grading a programming assignment laments that they do not understand how a student could have possibly come up with the answers that they did. Students do not comment their code enough, and the code itself is usually written in a language or style that is particularly obtuse. Plus, the turned in copy represents only a tiny fraction of the code that the student actually typed in while trying to make their project.

Would it not be nice, if you could follow a student's thought process along from beginning to end, and see not just the end product of their efforts, but all intermediate stages in between? Even better, if you could not only watch their code develop, but you could also know what they were thinking when they wrote it? That would give a teacher/grader much more information to work with in order to understand how a student developed their code, and more easily identify where they went wrong (and, where they had a great flash of insight!).

Consider a CS2 student working on a Java programming assignment to implement a linked list data structure. The student must define the appropriate Java class, but first thinks out load about what he needs to do.

> *≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫*

Then the student types in the class declaration:

> *≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫*
>
> ```
> public class LinkedList {
> }
> ```

The student next states out loud that they know one of the fields to add to the class:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
}
```

And, he then defines the value:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
}
```

At this point, the student wavers a bit. He's not sure how to complete the data structure.

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫

```
}
```

Fortunately for him, he knows his TA can read back what he's saying, so he puts a coded message in there for him.

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫
≪I guess I can leave it out for now and my T A will understand what I meant to write≫

```
}
```

The student then completes, to the best of his abilities, the functions in the LinkedList (`head` and `tail`) data structure and turns in his assignment to the TA.

At this point, the TA needs to grade this student's programming assignment. He runs the automatic testing suite, and finds that this student's program has failed all the tests. "How is that possible?" the TA thinks to himself. "He was doing OK in the beginning of the term."

Using Harmonia, the TA loads up the student's program. He reads the voice comments in the code and does not understand what the student meant to write. Perhaps if he played back the edit history of the document, he will be able to figure out where the student went wrong. Using Harmonia in XEmacs, the TA types in `M-x replay-history`. The document refreshes to an empty state, and replays each edit at a rate of one every three seconds. When the computer replays a voice comment, it plays back the audio of the comment in real-time synchronized with the text edits that occurred while the student was speaking.

Once the edit history plays back the last voice comment, the TA understands that the student knew he should extend his LinkedList with a pointer to the next LinkedList in the chain (or nil). The TA then adds his own voice comment to the student's code:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫
≪I guess I can leave it out for now and my T A will understand what I meant to write≫
≪Yes, I figured it out. You need to declare another field with the linked list type. Also try to think about what you would do to point to the end of the linked list.≫

```
}
```

The TA sends back the annotated assignment for the student to correct. He was able to provide the voice comment facility to give appropriate and more directed feedback to the student to help with his next revision.

2.2 Code Review

In this scenario, an employee of a networking startup in Silicon Valley is reviewing the code for a proxy server written by a colleague. He is reviewing the code at 2am because he is a night owl, and could not find a time to meet with his colleague that was suitable to both.

The code looks like this:

```
for (int i = 0; i < 10; i++ ) {
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

First, the employee states the obvious:

≪It's a loop of 10 connections.≫

```
for (int i = 0; i < 10; i++ ) {
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

Then, he notices an inefficiency. The employee's colleague is allocating a new Server object for every new connection when she should be reusing it:

≪It's a loop of 10 connections.≫

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

≪Why are you allocating a new server socket every time?≫

```
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

Reading onward, she notices an egregious security violation:

≪It's a loop of 10 connections.≫

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

≪Why are you allocating a new server socket every time?≫

```
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
```

≪@#@%! This is a huge security hole right here! You didn't check the input for validity before executing it.≫

```
  execute(input, i);
}
```

After making these comments, the employee quickly fires off two emails – one to the security officer at the company to turn off the alpha version of the server, and the second to his colleague berated her for leaving such as glaring security hole in their software.

This use of voice comments illustrates the benefit of informal voice commenting to annotate production source code and to rationalize quick decisions which everyone else can easily verify.

3 Implementation

The facilities for supporting voice comments are based on technology from the Harmonia research project [8] led by Prof. Susan L. Graham at the University of California, Berkeley. Harmonia is a language analysis library that can be plugged into an application to provide incremental lexing [18], parsing and semantic analysis services. For lexing, Harmonia supports flex-style lexers; for parsing, Harmonia supports LALR(1) and more commonly, a variant called GLR [16]. The GLR parsing algorithm is built upon LR(1) and adds the ability to parse grammars requiring any number of tokens of lookahead (infinite), as well as parse ambiguous grammar and retain ambiguities in the generated syntax tree.

For each language supported, Harmonia requires a language module plug-in, a shared library that contains the lexing tables, parsing tables, and syntax tree-walking semantic analysis code for a given programming language. A language module designer creates a *flex* description, a *ladle* (essentially bison + EBNF) LALR(1) or GLR grammar, and a set of *astdef* files describing an object-oriented walk through the syntax tree produced by the parser.

Harmonia is designed primarily as an analysis engine for text-based languages, but for the author's dissertation, he is exploring ways to use voice in the programming process and use Harmonia to support the process. This means generalizing the Harmonia framework to support lexers that don't analyze text, as well as parsers that can accept lexemes from multiple lexers.

3.1 Lexer Modifications

First, it was necessary to enable Harmonia's language modules to support more than one lexer table per language. It was relatively straightforward to change the lexer table pointer into a C++ STL vector of Lexer objects. However, it was a bit more complicated to extend the lexing algorithm to use a particular lexing table at any point in the lex process.

The incremental lexer is not a simple processor that transforms an input of characters into an output of lexemes. Its input is a stream of tokens that were previously lexed. When a file is first lexed, all of its characters are inserted into one giant, undifferentiated token. The incremental lexer may begin at any token in the token stream, at which point it decomposes the token into characters and feeds it into the lexer driver produced by flex. Tokens produced are stored in a token list to be incorporated back into the syntax tree by the parser. If more characters are needed to lex a token, tokens further ahead in the token stream are broken down and fed to the lexer. In order not to relex the entire input stream every time the lexer is invoked, the lexer compares the tokens produced to the tokens that it broke down. At the first point where these match (by a number of criteria), the lexer can stop lexing, since any further lexing will just create the same tokens.

It was necessary to convince the lexer to stop lexing when it reached a token that was produced by a different lexer table than the one the lexer was using. We created a mixin class, LexerChoiceMixin, which contained an integer to identify each unique lexer table. Each token type (each a C++ class) inherits from this mixin to record which lexer table produces the token. We then modified the incremental lexing algorithm to check the next lexeme to break down before passing it to the flex lexer. If this lexeme type was produced by a different lexer than the current one, it would treat the token as if it were an end of file marker. (Note, this also implies that the token preceding this *eof* would be limited to a lookahead of 0 characters.) The current lex would be stopped without harming the foreign token. Since that token, in all likelihood, would be need relexing itself, the incremental lexing algorithm would detect it, and switch its current lexer to the one found in the token, and then proceed to lex again. To restate the modification in one sentence, the lexer treats a token from a different lexer table (meaning different from the lexer is currently using) as an end of file and ends lexing. The algorithm will already continue the lexing process on the next token needing relexing without any changes.

Second, we created a new type of grammar terminal attribute, a VoiceComment. A VoiceComment is implemented as a LexerChoiceMixin whose lexer is number 1. (The default flex lexer is number 0). We added a new token type to the ladle grammar for Java, called "voicecomment", and gave it the VoiceComment attribute. We then wrote a method to create a voice comment token and insert it structurally into the parse tree. Finally, we created a new subclass of the lexer to correspond to lexer number 1. When the lexer switches to this lexer, it returns the voice comment (without destroying it as the

flex lexer does) and then indicates an end of file to force the lexer to switch back to the flex lexer.

3.2 Parser Modifications

In theory, the GLR parser doesn't need any modification to accept lexemes from a different lexer table. As long as the lexeme is described accurately by the grammar, everything should be fine. However, since the incremental parser drives the incremental lexer, and Harmonia is not as modular as is should be, a few modifications to the parser driver were necessary to get the parser in line. In addition, the history-based GLR error recovery mechanism [18] references character locations in the underlying token stream. This still needs to be modified to reference token position rather than character positions. Finally, there is a bug in the parser that causes it to infinitely recurse and insert thousands of copies of a voice comment into the parse tree. We will fix this soon.

3.3 User Model

We next designed the user model. We integrated the voice commenting feature into Harmonia-mode [7], our XEmacs plug-in. Harmonia-mode uses the features of Harmonia to support interactive error detection and display, syntax highlighting, indentation, structural navigation and selection, structurally-filtered searches, and elision throughout the code document.

Our desired user model would allow the user to talk modelessly and have their speech inserted into the document at various points as voice comments. Some design questions about the user model concerned us.

1. **If a user speaks a voice comment, where should it go?** Should it be inserted at the current cursor position? Perhaps we should speech-to-text the comment and use AI to interpret what they're talking about. For example, the user says "This field needs to be renamed." before a class definition. Since the user is talk about a field, the comment could be associated with the field declaration. Could a comment be associated not just with a text position in the buffer, but also with structural elements in the syntax tree? This way a comment could be associated a class, or a field, and even if that field is subsequently edited or moved, the comment could be kept near it. We decided to go with option 1, where the comment is inserted at the cursor position because it was the simplest option. AI is not our speciality, so we ruled out option 2. We ruled out option 3 as well because the text-oriented (as opposed to syntax-oriented) user model of the editor does not preserve enough information to determine anything but the text location where the comment should go.

2. **If a user speaks for a length a time while simultaneously editing the document, where does the comment go?** It could go at the position where they initially started speaking, or go at the end where they stopped speaking. Or the comment could span the entire range of characters. Unfortunately, Harmonia-mode associates text ranges (in XEmacs, these are called extents) in the buffer with nodes in the syntax tree. Without a node to represent this expanded range, we could not make the proper association. We decided to insert the comment at the cursor position where the user first began to speak, since that was likely close to what they intended to comment.

3. **When should comments be inserted in the code?** Should they go in as soon as the programmer stops speaking, or should we wait a few seconds? If the comments go as the user is coding, it could disrupt the visual flow of the program and interrupt the programmer's thought process – exactly the opposite of our intentions with this project. In addition, if the comment is too colorful for code (such as the expletive uttered by the code reviewer in the second scenario above), the speaker may not want it to go on. Likewise, the speaker may have forgotten to turn off the microphone when talking to a colleague in the room and the comment inadvertently was recorded. We could batch up spoken comments with their insertion locations and store these in a buffer. When enough time has passed or there are too many voice comments in the buffer, we interactively ask the programmer whether it should be inserted. It is important to present the voice comment inserted into the buffer with context above and below in order for the programmer to remember what they were thinking when they

spoke the comment.

4. **How should voice comments be rendered?** Should a voice comment be translated into the programming language's syntax for a comment? If yes, the user would not be able to visually identify which comments contained audio and which did not. We thus chose to present the voice comment in a distinct typeface. Should a voice comment be a simple one character glyph or should we present the entire speech-to-text translation with special delimiter glyphs? We chose to render the entire text of the comment in order to facilitate skimming. If all of the voice comments needed to be played back in order to find out what was in them, then one would have to listen to all of the comments to find anything. To reduce the clutter from the voice comments, we enable the user to elide their visual representation into a two character glyph (the boundary tokens of the fully expanded voice comment). The user can also click on the voice comment and choose to have the recorded audio spoken back out the speakers.

5. **Can voice comments be editable? If so, are they editable in text or in audio?** If voice comments are modifiable via text, it will be important to keep the audio in sync. One would have to know the time indices of all of the words in the audio stream and be able to cut and paste them. If the user reordered the words in the comment, the audio could be recut to match. But, what if a user deleted a few characters of an existing word, or even added a completely new word? Should the tool synthesize new speech and insert it in the audio? Since this project was more about how the voice comments would be used as an annotation tool, and not intended to be a complete prototype, we prohibited all editing of voice comments.

We made all the aforementioned modifications to our harmonia-mode for XEmacs (all except that code comments are automatically inserted into the code buffer two seconds after the programmer has finished speaking each one) and informally tried out our prototype. Barring some initial technical difficulties with IBM ViaVoice related to discovering when the user has started and stopped speaking into the microphone (the speech recognition engine only reports when it is decoding voice into text, and reports the overall input volume level, which the author used to infer when the programmer stopped speaking), the prototype worked well.

4 Experiments

We plan to conduct several experiments to see how voice comments can be used by both novice and expert programmers to create better comments in their code. For novices, we would like to deploy the system to three students in an introductory programming class (conducted in the Java programming language), and ask them to use voice commenting on one of their programming assignments. We hope to see an increase in commenting in the program itself as compared to the rest of the students in the class.

Our main metrics will be the total number of comments, number of comments per class, field, method and line in the code, as well as the number of characters in the comments themselves. Another metric will be the number of comments that are about semantic information in the program, rather lexical and syntactic. This will be mainly a comparative one with the other students in the class, since novices tend to comment poorly anyway. Indirect metrics will be evaluations from the students as to the distraction or benefits they see from talking about their code, and an evaluation from readers who grade the programming assignments to see if they feel that they gain a better understanding of what the students were thinking when they were writing their programs. This last evaluation is critical to understanding whether or not playback of the comment audio is useful to the reader, as well as if it is possible to wade through the (hopefully) copious quantity of comments in the code to find out what is important.

For experts, we can conduct a similar experiment except that instead of a programming assignment, the expert's current project will be studied. A tool created by another member of this Software Engineering class can report statistics on commenting behavior found in a code repository [12]. We plan to use this to establish a baseline for an expert's tendency to comment their code. Then, we encourage the expert to use our voice commenting system, and look at the changes, if any, in their commenting behavior by examining the repository. We can use the same metrics as for the novices as well as the same evaluation by the programmer to

identify any cognitive issues that interfere with the programming activity.

5 Future Work

The experiments described above have not yet been completed. The author plans to conduct them over the next few months. In addition, the user interface that enables a programmer to interactively insert their voice comments en masse into the document has not yet been completed either. This will also be completed in the next few months. Playback of the audio portion of a voice comment is also be forthcoming.

6 Conclusion

This paper describes a novel system for improving the quality and quantity of comments in programs. By using their voice to comment their code, programmers can exploit an additional input channel that should not interfere with their ability to code at the same time. Using the Harmonia framework to support program code analysis and structural editing, it is relatively straightforward to add this new form of commenting to any programming language.

We hope our experiments will show that programmers, both novices and experts, will be able to use this tool to comment their code more completely and descriptively. Exploration of the cognitive interference issues will be critical to its success – speaking about one's code may interfere with programming, or even if not, the stream of conscious style of comment may increase the likelihood that the comments will be lexical or syntactic in nature, rather than the more useful semantic form of comments.

Finally, the one incontrovertible benefit of voice comment is that it will form an indelible and more complete record of the programmer's process while performing his duties. Programmers have a lot to say – it is time that we started capturing it.

7 Acknowledgments

The author would like to thank Michael Toomim for creating the modern version of harmonia-mode for XEmacs as well as making many of the changes necessary to render and edit voice comments. Marat Boshernitsan should also be thanked for his assistance in modifying the Harmonia lexer and parser to accept voice comments without crashing.

References

[1] Daniel Brantley and David Dillard. Software tools in the service of documentation. In *Third International Conference on Systems Documentation*, pages 60–70, 1984.

[2] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. Principles for designing programming exercises to minimise poor learning behaviours in students. In *Proceedings of the on Australasian computing education conference*, pages 26–33. ACM Press, 2000.

[3] Françoise Detienne. *Software Design – Cognitive Aspects*. Springer, 2001.

[4] Timothy E. Erickson. An automated fortran documenter. In *Proceedings of the international conference on systems documentation*, pages 40–45, 1982.

[5] Michael Ericsson, Magnus Baurn, Jonas Lwgren, and Yvonne Wrn. A study of commenting agents as design support. In *Proceedings of the conference on CHI 98 summary : human factors in computing systems*, pages 225–226. ACM Press, 1998.

[6] R. Escalona. Case study of the methodology of j. d. warnier to design structured programs as systems documentation. In *Third International Conference on Systems Documentation*, pages 95–100, 1984.

[7] Harmonia Research Group. *Harmonia-Mode Project Documentation*. http://acacia.cs.berkeley.edu:8081/harmonia2/projects/harmonia-mode/index.mhtml.

[8] Harmonia Research Group. *Harmonia Research Group Home Page*. http://www.cs.berkeley.edu/ harmonia.

[9] IBM, Inc. *IBM ViaVoice Product Home Page*. http://www-4.ibm.com/software/speech/.

[10] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings on the seventeenth annual international conference on Computer documentation*, pages 147–153. ACM Press, 1999.

[11] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. In *Proceedings of the Third ACM SIGSOFT '88 Symposium on Software Development Environments*, pages 173–182, November

1988. Published as SIGSOFT Software Engineering Notes, volume 13, number 5.

[12] David Marin. Searching source code to enable code reuse, 2002. UCB CS294-1 Class Project.

[13] Gregory R McArthur. If writers can't program and programmers can't write, who's writing user documentation? In *Proceedings of the Fourth International Conference on Systems documentation*, pages 62–70. ACM Press, 1985.

[14] R. Douglas Riecken, Jurgen Koenemann-Belliveau, and Scott P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 177–195, 1991.

[15] David Roach, Hal Berghel, and John R. Talburt. An interactive source commenter for prolog programs. In *Proceedings of the conference on SIGDOC '90*, pages 141–145. ACM Press, 1990.

[16] Masaru Tomita. *Efficient Parsing for Natural Language — A Fast Algorithm for Practical Systems*. Int. Series in Engineering and Computer Science. Kluwer, Hingham, MA, 1986.

[17] Vicente Lopez Trueba, Julio Cesar Leon Carrillo, Oscar Olvera Posadas, and Carlos Ortega Hurtado. A system for automatic cobol program documentation. In *Third International Conference on Systems Documentation*, pages 36–43, 1984.

[18] Tim A. Wagner. Practical algorithms for incremental software development environments. Technical Report CSD-97-946, University of California, Berkeley, March 11, 1998. ftp://sunsite.berkeley.edu/pub/techreps/CSD-97-946.html.

[19] J. M. Yohe. An overview of programming practices. *ACM Computing Surveys (CSUR)*, 6(4):221–245, 1974.