Voice Notes:
A Discourse Analysis of Programmer Voice Annotations

Jinger Zhao

TR-18-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

VoiceNotes: A Discourse Analysis of Programmer Voice Annotations

A Thesis presented

by

Jinger Zhao

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 6, 2004

Abstract

Poor program documentation makes code hard to understand and maintain. Studies on pair programming and software documentation suggest that enabling programmers to annotate their programs by voice may improve program documentation and confer some of the benefits of pair programming to single programmers. To design a voice annotation tool, however, programmers' voice commenting behavior must be understood. This thesis introduces the VoiceNotes system, a plugin for the Eclipse Platform that captures programmers' speech during programming and provides a playback mechanism for recorded speech and text. A discourse theoretic method of analyzing programmer voice comments is developed and applied to an empirical study of programming recorded using VoiceNotes. The results show that a predictable time relation exists between comments of different intentional structures and their related coding actions. The analysis also suggests that the commenting behavior a voice annotation tool can elicit from single programmers contains many of the programming-enhancing phenomena that have been found in pair programming studies. The implications of these findings for the design a voice annotation tool and directions for further study of programmer voice commenting behavior are discussed.

1 Introduction

Documentation of programs is notoriously sparse and uninformative. Programmers testify that they seldom put effort into documenting because they find explaining code that they understand boring and because they must divert their attention from coding in order to write documentation. However, documentation is crucial for the transfer of knowledge about code between programmers, and, for reasons discussed later, an automatic documentation generator is unlikely to leave information as rich as can the code's author. Thus, a better way of encouraging and recording programmer documentation needs to be developed.

One way to improve documentation is to enable programmers to document their code in a way that does not interfere with their normal programming activity. Speech is the most promising documentation modality to explore: programmers already speak as they code and do not have to stop typing code while they are speaking. Furthermore, studies suggest that programmers are willing to leave more information about their code, including their assumptions, the code's intended use, and the tradeoffs they considered, when they speak [23]. In pair programming, for example, when programmers speak about their code to their partners, they talk about important decisions they have made in their code that they typically do not record in any documents [25].

Recording speech from pair programming is easy: programmers speak naturally as part of the programming process. Despite the increasing popularity of pair programming, most programmers still work alone and do not have an audience to speak to. To elicit verbal comments from single programmers, we could create a *software collaborator* to play the role of the missing human partner. One goal of such a software collaborator is to induce single programmers to speak about their code and to capture these statements as documentation. To be a good replacement for a human partner, the software collaborator needs to respond intelligently to the verbal comments that programmers make and help programmers attach their comments to their code. A secondary goal of the software collaborator would be to help programmers plan and understand their code better by making the programmers articulate their ideas about what they are writing.

This thesis describes the first step toward building such a collaborator: understanding what kinds of verbal documentation programmers leave and how they leave it. Earlier work has not formally or rigorously analyzed programmers' voice commenting behavior, nor has the content of voice comments been extensively studied or compared to textual documentation. Thus, we created VoiceNotes, an experimental tool for the capture and analysis of programmer voice commenting. VoiceNotes's sound and text manipulation modules are designed to be

reusable for a software collaborator that helps the programmer manage voice comments.

Using the VoiceNotes recorder, we recorded five programmers who were instructed to leave documentation so that people modifying their code later can understand it. The instructions that were given are ones they could expect in a workplace or in an academic project. We analyzed the programmers' voice comments according to the theory of tripartite discourse structure developed by Grosz & Sidner [12]: In particular, we used the linguistic structure of the comments to break the recordings into comment units (discourse segments) and labelled each unit with respect to its intentional structure and attentional state.

We used both pauses and topic changes between utterances to determine the boundaries between comment units. The results of linguistic segmentation show that commenting frequency varies greatly from subject to subject and that programmers dictate their own coding verbally to varying degrees. These variations between programmers complicate the process of detecting comment boundaries difficult for voice annotation systems. Another important challenge we identified for automatic documentation systems is the phenomenon of audio cross-referencing: subjects often referred to comments they made earlier or had not yet made in their explanations (Section 5.2.2).

In accordance with earlier studies [28], we found that subjects most

often focus their attention on functions. Because it was difficult to identify the attentional state of complex comments, a procedural classification scheme to replace attentional categories is suggested. When we labelled subjects' comments with respect to their intentional state, on the other hand, we discovered a rich structure that is at once informative and tractable. Subjects overwhelmingly prefer speaking about what they intend a piece of code to do immediately before writing the relevant code. Rarely do they go back to explain what they have written afterwards. Moreover, programmers' voice commenting behavior exhibits predictable acoustic and temporal patterns which a software programming collaborator could take advantage of.

Most importantly, the combination of a voice annotation recorder and the program documentation instructions we used successfully elicited speech from single programmers. Furthermore, the speech we captured contains information that is much more in-depth than that usually found in textual documentation. The single programmers' comments to VoiceNotes are similar to what pair programmers say to each other. These results suggest that programmers would interact with software collaborators to manipulate voice documentation. Furthermore, single programmers may get some of the same cognitive benefits from voice annotation that pair programmers get from their conversations during programming.

The rest of the sections in this thesis are organized as follows: Section 2 explores earlier work on voice annotation systems and pair programming studies and introduces open questions that these studies raise about programmers' voice annotations and the design of voice annotation tools. Section 3 describes the architecture of VoiceNotes, the voice annotation platform we have developed for recording programmer vocal and typing behavior and playing back programmer verbal comments in real-time synchrony with the program edit history. Sections 4, 5, and 6 present the methodology and results of a user study based on the VoiceNotes recorder and playback tools and develop an approach to analyzing programmer voice annotations and modelling commenting style. Finally, Section 7 describes the limitations of this study, and Sections 8 and 9 suggest future studies and applications for the VoiceNotes voice annotation platform.

2 Background and Motivation

As software systems have gotten much larger and older and programs have become more complex, programmers' need for tools to help them understand, remember, and maintain code have become more urgent. Currently, textual documentation is the primary way programmers share knowledge about the structures and procedures in complex programs, but the textual documentation programmers leave is notoriously

insufficient in both quantity and quality. Programmers prefer to spend their time solving new problems rather than explaining existing code, and they expect other programmers to be able to figure out their code without much guidance. The Joel on Software Forum at Fog Creek Software conducted a survey on "Why Do Programmers Hate Documenting?" [1] [15]. According to one respondent:

> Programming is a largely a creative, problem-solving effort. Documenting is largely a teaching and communication effort. We like programming because we like creating solutions to problems. (Some of us like finding the solution so much that once we've figured out the solution we're not interested in implementing it!) We don't like documenting because we feel like we're not really solving a problem, we're just describing something.
>
> Also, I think a lot of programmers view documentation as largely being a waste of time and effort. When you are writing the equivalent of, To create a new document click 'File', then 'New', the thought going through your head is what a colossal waste of your time this is. After all, anyone can figure out how to do simple things, right?
>
> I don't believe the claim that programmers are lazy. I do believe that programmers dislike doing things that are not programming. Like writing documentation, working on schedules, defect tracking. And we'll procrastinate on those things, or do the minimal effort necessary.

This commentary suggests three roots of poor documentation: First, textual documentation requires programmers to shift from "problem-solving" and "creating" to "teaching" and "explaining." Typing doc-

[1] Appendix C presents more programmer testimonials from the Joel on Software Forum.

umentation conflicts with the implementation process: programmers must break concentration and motor activity away from their coding to think and write in prose [3]. It has been shown that typing comments imposes a larger mental load than creating documentation by another avenue of input, such as voice [28].

Documentation is also time-consuming. The programmer must essentially program three times: once in pseudocode, once in code and once again in prose. Because programmers are not directly rewarded for their documentation skills, nor do the majority enjoy documenting [20], programmers spend as little time commenting as possible. The resulting comments are few in number and lacking in detail.

The third reason programmers leave poor documentation is that they assume that, with enough time, any code they write can be deciphered by other programmers. Thus, they view comments that explain the code as redundant and unnecessary. However, as another programmer on the discussion forum noted,

> Of course you shouldn't comment every line of code. In fact, for most methods you only need a single comment describing the intended use and parameters. But any real-world system will have plenty of cases where explanatory comments are helpful and greatly reduce the time spent maintaining the code, especially by reducing bugs introduced by the maintainers through misunderstanding.

That is, a record of programmers' *intentions* and the *assumptions* clarifies not only how the code operates, but also how the programmers inheriting a program should use the code. Knowing how existing code is intended to be called is especially important to programmers creating ever-larger software systems. The problem of poor documentation intensifies when a programmer has worked on many systems or is working in a group of programmers that must extract sense from, modify, and maintain each other's code.

Managing the complexity of software is crucial, and current systems of documentation, programming tools and languages do not make documentation easy enough for programmers. Programming languages constructs are not lucid enough to be their own documentation [22]. Although automatic documentation systems such as GETREF and Javadoc have been developed to help programmers create textual comments, such tools only serve to publish APIs and cannot explain program logic and design [5, 17]. In these systems, documentation is primarily written after code has already been completed and usually only records the steps the code takes to compute a certain value or perform a specific task. Although comments about functionality help programmers understand code, they do not include details about why the code was written the way it was and what it was intended to do.

One way to help programmers understand code without good documentation is to eliminate their dependence on documentation entirely. So-called "posterior" approaches have been developed to generate descriptions of and provide information about code without guidance from the code's authors. Posterior methods include textual, lexical, syntactic, control flow, and data flow analysis, program dependence graphs, and cliche recognition [6]. Posterior analysis is not satisfactory for two reasons: a) it cannot comprehensively describe program functionality, and b) it does not capture how the code's authors intended the code to work nor how they intended the code to be used.

If we could create a unobtrusive documentation system that encourages programmers to document in a way that is more compatible with their usual programming habits, they may leave more frequent and informative comments. Previous studies (Section 2.1) suggest that speech provides a strong alternative source of program documentation. Any observation of programmers over a brief period of time reveals that the comments that programmers make in discussions with their coworkers or with themselves contain valuable information about the assumptions, requirements, and intentions of the program and directions on how to change it. Particularly for collaborative projects involving many people, such comments about the decisions and assumptions in code are as important to designing and maintaining a working, robust program,

as the lines of code themselves.

A documentation system can exploit the fact that people already "comment" when they talk to themselves or to coworkers. However, programmers do not always have a human audience present for whom to articulate their thoughts. If we create a program that encourages, captures, and responds to programmers' verbal comments much information can be captured for documentation. Such an agent for programmers would act as a *software collaborator* that enables programmers to record documentation easily, organizes their verbal comments easily, and provides an audience for the comments. Like a human partner, the software collaborator would respond intelligently to mistakes and other actions and work with programmers in the way that collaborative interfaces like GLIDE, DIAL, and WAID work with their respective users [1, 24, 26].

A key component of such a complex agent is a voice annotation system that provides for programmer voice interaction with the computer. An interactive voice annotation system offers many benefits in addition to increasing documentation. First, voice carries more information than text alone. Voice has texture, tone, and pitch and thus contains inflectional as well as lexical information. Multimodal input and emotion have both been shown to increase the memorability of associated objects [9, 11]. Thus, verbal documentation may be easier to remember

and understand than written comments.

The goal of the work described in this thesis is to capture and analyze programmer voice commenting behavior in order to provide the basis of the computer programming collaborator described above. The next section examines earlier work done on voice commenting systems for programmers and describes how the voice commenting behavior study we present builds on previous research.

2.1 Related Work in Voice Annotation

At least three earlier efforts have been made to create voice annotation tools for programmers: Variorum, the Verbal Source Code Descriptor, and the Software Concordance project [6, 7, 28]. The first of the projects, Variorum, tries to take advantage of the "code walk-through" session, a standard procedure in most software companies, by recording the drawings and vocal descriptions made during the walk-through using electronic pen and paper and microphone. These drawings and voice files are associated with a static location in the source code being described and stored separately in annotation files containing interleaved voice and graphic data. The Variorum system combines annotation files and source code in a hypertext browser that displays the transcriptions of subjects' voice comments and drawings as links above the code segments to which they refer.

The Variorum system assumes that programmers insert all comments *after* they write the relevant code. The Variorum system thus treats annotations as a form of post-facto review rather than information to be captured by an interactive tool integrated into the development process. Furthermore, Variorum retains no information about the intermediate states of programs nor about the discarded designs of program structures and segments. Such information would not only inform programmers working on the code later about the purpose of existing structures, but it would also warn them against wasting time exploring codepaths that have already been found to break. Also, Variorum's annotations cannot be modified, nor can the program locations with which they are associated change. Indeed, Variorum was created and is useful only for programs that are relatively stable.

Although the authors of Variorum did not perform extensive user studies on their system, they did find that speech is the preferred medium of documentation when programmers are given a choice between speaking, drawing, and typing their comments. They measured the percentage of annotation time[2] for spoken, drawn, and typed documentation for three algorithm-intensive, short programming problems. They found that for the suffix tree building problem, 90.94% of the annotation time was speech, and spoken comments took 87.95% of the

[2]The "annotation time" is the time required to play back the recorded annotation.

annotation time for Fibonacci heap sort and 89.39% of the annotation time for the radix sort problem. As one subject testified, "The audio documentation helps to clarify the data structure and program flow." Thus, speech seems to be an important medium of information retrieval and gathering.

The Verbal Source Code Descriptor [28] and the Software Concordance Project [6] both allow programmers to make voice annotations as they code. The VSCD transcribes vocal comments to text in a separate code-documentation window, the contents of which are kept in sync with the source code the user is commenting. Although the authors showed that their system slightly eased the mental load of documentation and increased users' commenting frequency, they did not explore the content difference between text and verbal documentation. Like Variorum, the VSCD depends on users to update annotation content and location; they provided a speech-to-text commenting system and assumed no difference between the types of annotations programmers would speak and those they would type. On the contrary, comparisons of text and spoken comments have shown that *qualitative* as well as quantitative differences between the two annotation modalities are key advantages of voice annotation during and after program development [23].

Instead of transcribing comments, the Software Concordance Project

(SCP) incorporates multimedia documentation into a web of hyperlinked program and media documents, including voice annotation sound files, that create a "collection of documents hyperlinked into a literate whole." The project's goal is to analyze and maintain semantic relationships between parts of a program, they provide a rich browsing but not editing experience. Voice and graphic documentation must be created in separate files and then linked to the relevant parts of code by hand, and any updates to the links and annotations must be made manually by the programmer. Because the SCP depends on users creating external documents outside of code development, it is it easy for programmers *not* to use the system. Thus, the tool actually adds to the existing burden of program documentation. Furthermore, no user studies have been done to prove that the SCP helps programmers leave better documentation and lead by the stairs.

As Slaney *et al.* note, "To design interfaces that effectively support human-computer interaction, we must first understand the complex behavior computer users exhibit when carrying out their jobs" [27]. Earlier voice annotation tools enforce certain commenting procedures without first analyzing programmer verbal comments. To develop a complete voice annotation tool with an intuitive collaborative interface, however, we must know the programmer actions and dynamics that such a system must handle.

User studies of programmer voice commenting behavior have been sparse. The most extensive of these, Soudian & Fels's study of programmers' usage of the VSCD [28], only characterized how comments were distributed across different types of program structures. They were able to describe comments in this way because the VSCD, on which the study was based, asked programmers to indicate a location in the code to which to attach each of their comments, and Soudian & Fels labelled comments according to the most local program structure. However, programmers often speak about structures that are not currently visible in the editor, about the relationship between noncontiguous structures, and about code that no longer exists or does not yet exist in the program. Because Soudian & Fels's study and results were so closely tied to how the VSCD works, they were not able to analyze the full spectrum of programmer voice annotation behaviors. Soudian & Fels did not explore the time correlation between programmers' comments and their corresponding coding actions. The ability to calculate and predict temporal relationships between the incoming sound stream and programmer edit actions is essential to a tool that encourages, records, and organizes voice annotation during program development.

The most elucidating research on voice commenting of programs comes from the non-programming domain: Stifelman's Audio Note-

book presents the best non-programming analogy for programmer voice annotation studies [29]. Stifelman's goal was to correlate lecture audio recordings with notes written during the lectures. Notetaking, Stifelman posits, can be distracting to students and may decrease when the material is very complex or unfamiliar to the listener. She wanted to free listeners from taking detailed notes so that they could devote more attention to the speaker. Stifelman created a notebook that stores lecture audio time-synchronized with written notes. The notes became structural indices into the audio recording. These indices helped students find the relevant portions of audio as they reviewed their notes. To make the most informative indexing possible, Stifelman developed a method based on discourse theory for processing and aligning the sound input data from the lecture audio with the notes that students take during class. This alignment allowed students to locate lecture segments that correspond to a given location in their notes.

Similarly, we correlate the speech of programmers with their coding. Typing documentation can be distracting to some programmers, just as notetaking can distract students. A goal of voice annotation is to free programmers to devote more attention to the program, so they are not always required to write detailed documentation. Programmers' coding actions can serve as structural indices into the audio comment stream, and these indices can inform the correlation of programmers'

utterances with coding tasks.

There are important differences, however, between the world of lecture audio and that of program commenting. Stifelman's domain involves only a two-dimensional correlation between notetaking actions and the audio input. In program voice annotation, however, editing actions result in deletions, movements, and insertions. Although the audio may correspond very well to the coding actions, these actions do not grow the program linearly. Many actions are cancelled by later actions, and many comments made during the coding process may later become irrelevant or have no point of attachment in the code. Program edits are also noncontiguous: different components of the same structure are created at different points in the edit history. In notetaking, in contrast, the length of the notes increase monotonically, with minimal back-reference and comparatively little erasure. The notetaking and lecture audio history thus correspond nearly linearly with the resulting pages of notes.

Despite these differences, two key lessons can be taken from Stifelman's techniques and approach. First, Stifelman found that locating phrase boundaries in the recorded lecture audio and then allowing users to navigate from phrase to phrase resulted in a better user experience than having users review audio notes with rewind and fast forward buttons. Thus, we will also analyze programmer audio at the phrase

level.

Second, a relationship may exist between the time the audio is recorded and the time the corresponding writing or programming action is done. Stifelman found that students take notes a brief period of time after they hear the corresponding audio, because they must listen to the lecture before writing down a record of what they heard. We identified such time patterns in programmer commenting and extend Soudian & Fels's analysis of programmer comments to the temporal domain.

Thus, it is clear from an examination of earlier work in programmer voice commenting that the system of programmer voice annotation capture and the discourse theoretic analysis of voice commenting we develop in this paper represent a novel approach to helping programmers understand and create code. Instead of manipulating text documentation or allowing programmers to create text documentation by speaking, we want to introduce programmers to untranscribed voice annotation as a new documentation modality. To develop a voice annotation system, we need to understand how programmers speak about their code to determine whether and how we can take advantage of speech, the untapped source of rich documentation, with software.

The specific issues and goals addressed in this thesis are:

- Building a recording and playback mechanism for programmer voice annotations.

- Determining whether programmers can be motivated to use voice commenting.
- Identifying the types of voice documentation programmers leave if they had a voice annotation tool.
- Developing and applying an approach to analyzing programmer voice annotations.
- Determining the features and technical foundations a voice annotation tool needs.

3 Architecture: the VoiceNotes System

We built the VoiceNotes system as a plugin on top of the Eclipse platform to gather voice annotations from programmers while they code. The VoiceNotes voice annotation engine extends the Eclipse Java Development Toolkit. Appendix G contains the details of the VoiceNotes plugin architecture.

The two main components of VoiceNotes are the recorder and the playback tool. Programmers interact with the recorder while they are coding, and annotators use the playback tool to analyze the recorded data. All controls for the recorder and playback are displayed in the VoiceNotes Viewer, shown in Figure 1. These components are discussed in more detail below.

3.1 VoiceNotes Recorder

We tried to keep the recording interface lightweight and unobtrusive to programmers, to collect and analyze behavior that is as close to natural as possible. The programmer presses the RECORD button to begin recording sound and presses the STOP button to stop the recording. The Recorder associates each voice comment file with a history of *highlighted regions*, taken from the Eclipse Java editor. The highlighted region is the stretch of code enclosed by braces within which the cursor lies. For example, in Figure 2, the cursor lies within the `findNode` function, causing the Eclipse environment to highlight the entire `findNode` method listing. The highlighted region gives a rough estimate of the program structure the programmer is editing at any given time. We use it as an indicator for the program structure on which the programmer's attention is focused.

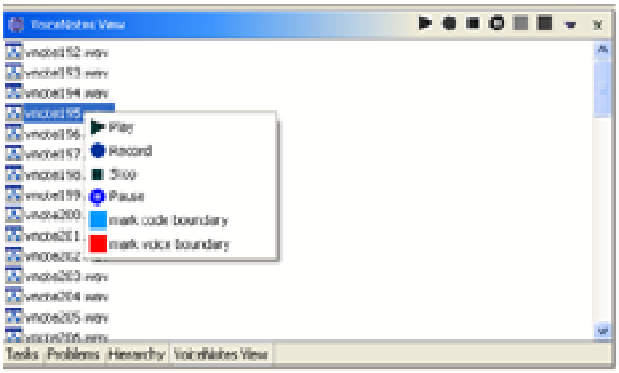

Figure 1: A screenshot of the VoiceNotes viewer in the Eclipse Java development environment.

```
public Graph() {
}

int findNode(Object aNode) {
    int i;
    for (i=0; i<Nodes.size(); i++) {
        if(Nodes.elementAt(i)==aNode) return i;
    }
    return -1;
}

int findEdge(String edgeLabel, Object toNode, Object fromNode)
    int i;
    for (i=0; i<Edges.size(); i++) {
```

Figure 2: The highlighted region is indicated by the blue highlighting along the left vertical ruler. The cursor is between the 'o' and the 'd' in "Nodes" in the **for** loop's conditional. The smallest program structure delineated by braces that encloses the cursor is the **findNode** method.

3.2 VoiceNotes Playback Tool

The VoiceNotes playback tool was created to enable annotators to review the sound and code recordings of programmers. The tool enables the real-time analysis of the correspondence, if any, between speech and edit actions. Any programmer interested in the history of the development of the program can use the playback tool to review the coding and speaking history.

The playback tool interprets the log files left by the recorder. The annotator can initiate playback by double-clicking on a voice annotation file name or by pressing the play button. When playback is triggered, the selected sound file is played, and the corresponding highlighted regions are displayed in Eclipse's console viewer. The annotator can thus watch and listen to program development in real time.

4 Experimental Design

The two goals of our user study were, first, to determine whether programmers can be motivated to leave voice documentation, when they are given the ability and motivation to record their comments vocally, and, second, to analyze what programmers say and how they say it. We conducted a user study with single programmers to analyze more precisely the correlation between the voice comments and the code they

produce.

4.1 The Programming Task

Subjects were given an hour to implement a simple Java API for a class representation of a multigraph (Appendix D gives the task instructions and Appendix E the API listing). Subjects completed the task using the Eclipse Java Development Environment and had access to the internet as they were programming. No one else was in the room. Subjects spoke their comments into a Altec Lansing headset with microphone, model AHS202. Although subjects could pause the recording during the hour, they were recorded continuously throughout the programming session so that we could capture and analyze all of their speech.

This particular task was chosen because it is short and manageable for intermediate programmers to complete in an hour, but also gives programmers the freedom to choose their own internal representations of the multigraph and to make algorithmic decisions based on their own assumptions of how their class will be used. The API was given to programmers without any comments in the code, in case existing comments would cause them to follow a particular commenting convention.

To motivate programmers to employ the voice commenting system as they would a vocal annotation tool, we gave the verbal instructions below in addition to the written instructions in Appendix D:

> You will now implement a simple API in Java. As you are coding, please verbally comment your code. The sound recording of your comments and a transcript of your speech will be saved for the subjects in a subsequent phase of our experiments to listen to and to read, respectively. They will need to understand your implementation as well as possible, as they will be extending and changing it. They will need to know what you are doing and why you are doing it in order to succeed in their experiment implementation.

These instructions contain motivations to comment code similar to those of software development in the real world. Programmers are often asked to leave comments for other programmers who will "need to understand [the] implementation" and "extend" and "change" it. The instructions also avoid specifying when and where to comment and how the comments will be evaluated, thus allowing programmers freedom to comment as they deem appropriate to the task and circumstances. Subjects were neither encouraged to type comments into the code, nor prohibited from textually commenting their programs. The analyses of these textual comments are saved for subsequent research.

Five subjects participated in this study: four Harvard undergraduates and one professional Java programmer. They were given the Subject Questionnaire in Appendix B to determine their experience with programming languages and tools. Different levels and types of experience (software engineering vs. academic coursework) may affect

Subject ID	1	2	3	4	5
PLs Known					
C/C++	1,3	3	0,1,2,3,	1,3	2
Java	1,2	1	0,1,2	1	2
Lisp/ML	1	1	0,1,2,3	x	1
Prolog	1	x	1	x	1
Perl/Scripting	1,2,3	x	0,1,2,3	x	2
SQL	1	2	0,3	x	2
Most Frequent	C/Perl	C/C++	Python	C#	Java
Programming Experience	5	8	8	7	8
Years Programmed	5	4	4.5	4	13
Java Experience	3	4	5	2	9
Years Java	1	1	4	1	7
Programming Environments					
Visual Studio	1	3	0,3	0,1,2,3	x
Eclipse	2	x	x	x	x
Emacs	x	x	0,1,2	x	x
vim or another text editor	1,2,3	2	0,1,2,3	1,2	2

Table 1: Subject responses to Subject Questionnaire. The numbers are coded as follows: 0 - used for personal side projects, x - no experience, 1 - used in class on homework assignments, 2 - used in class on a large system, 3 - used in a job or internship. Programming Experience and Java Experience are rated on a scale from 1 (inexperienced) to 10 (very experienced).

the frequency, quality, and overall style of comments[3]. The subjects' responses are given in Table 1.

Because the task required familiarity with elementary graph theory, we asked all subjects whether they were comfortable with the definitions of the functions in the API they were to implement before recording them. [4]

[3] A definition of commenting style is suggested in Section 6.4.4.

[4] "What is a Hamiltonian cycle?" was the most common question, followed by, "Do you have a copy of CLR?"

5 Results

We wanted to answer three specific questions about the programmers' comments:

- How can voice comments be identified (by people and by automated systems)?
- What parts of the code do the voice comments refer to?
- What purpose did the programmer have for making different kinds comments?

These questions correspond to the three structures that compose discourse, according to the tripartite theory developed by Grosz & Sidner [12]. The three structures are

- Linguistic: The division of the audio stream into segments based on cue phrases and pauses.
- Attentional: The entities (of the task) the speaker is focused on.
- Intentional: The purposes and goals of the speaker.

Thus, we first break up the audio records of each programmer into discourse segments according to the linguistic structure and then label each segment with an attentional state and an intentional category. Each of these procedures is described in detail below.

5.1 Linguistic Structure: Comment Unit Segmentation

To describe the relationship between the sound and edit history, we must first define what we are trying to correlate in the sound and program development history. Possible correlation units for sound files are words, phrases, and sentences, and possible correlation units for the program edit history are individual keystrokes, task segments, and durations of continued typing.

Several of these possibilities may be discarded immediately. Correlating individual words and keystrokes is not an interesting or useful first analysis of programmers' voice comments: our goal is to record and associate full comments with the editing tasks to which they refer. Grosz & Sidner divide discourses into discourse segments. They used cue phrases and changes in attentional and intentional state to identify discourse segments in their text dialogues. We use programmers' vocal behavior and comment content to break programmer recordings into analogous segments, which we call *comment units*. After reviewing a few subjects' sound recordings, we found that subjects would pause between 3 and 10 seconds when switching topics or changing to the implementation of a new section of code. Thus, we developed the rules below for breaking the sound files into comment units. Each comment unit is defined by a start timestamp, the time in seconds of the first syllable of the first word, and an end timestamp, the time in seconds

of the final syllable of the final word.

1. **Topic segmentation**

   Separate utterances are considered separate comment units if they talk about different topics. For example, Subject 2 stated as he finished his `removeNode` function, "In the future I can imagine writing a compact function that will compact all the nodes," and followed it with, "So now I'm going to write a vector object." These two statements refer to a potential extension of the class's functionality and a vector Subject 2 is about to implement, respectively. These are two different topics and are thus recorded as different comments.

2. **Time segmentation**

   Any two speaking durations with more than five seconds in between are separate comment units.

3. **Interruption**

   A duration of speaking may be divided into two or more comments if the subject interrupts him or herself in the same utterance. For example, as Subject 4 began to define an adjacency matrix, he said, "so we start out by creating a new matrix and setting temp to point to the old one," but he stopped typing when he recalled a prerequisite subtask: "oh I just remembered I have to go back to

the addNode function and increment the counter." The first half of his comment explains the function and manipulation of the `temp` variable. The second part refers to an unrelated subtask to which the subject diverted his attention and which caused the subject to navigate away from his matrix-creating code.

It is possible that a subject talks about the workings of one method over several minutes and sentences, during which the subject may get distracted for more than five seconds by reading documentation or perusing something in their code. The rules would divide such a stream of utterances into several comment units in succession when arguably only one coherent comment was spoken. Section 8 discusses ways to improve the algorithm to account for comment units with long pauses.

The amount of time each subject spent talking was not calculated and is expected to vary directly with the number of comments. Although the average length of time each comment takes could be useful to a voice annotation tool that needs to determine comment unit boundaries automatically, the amount of talking time is not meaningful in a discourse analysis of comments.

Figure 4 shows the total number of comment units for each subject. There does not appear to be a strong correlation between any programming experience metric recorded in the Subject Questionnaire and the total number of comments made.

The subject recordings were reviewed in detail using the VoiceNotes playback tool and manually transcribed. Examples of comments from the transcriptions are given in the analyses below.

5.2 Comment Unit Labelling

5.2.1 Attentional Structure

According to Grosz & Sidner, attentional state is defined by a set of "focus spaces," which are manipulated by a process called "focusing." The act of focusing "associates a focus space with each discourse segment; this space contains those entities that are salient—either because they have been mentioned explicitly in the segment or because they became salient in the process of producing or comprehending the utterances in the segment." Forming these focus space associations is the process of determining attentional structure.

It follows from the nature of the programming task that the focus spaces in programming discourse are often defined by program components. Because users attend to editable units of programs, the structures that make up the programs, we define attentional categories based on the programming language's hierarchical structures. In our categorization scheme, the focus structure, the set of focus spaces available, is equivalent to the hierarchy of program structures that a programmer can attend to, compose, and manipulate.

Soudian-Fels Attentional Categories	Corresponding Attentional States in This Study
Overall approach	Overall approach
File header	Class
Function header	Method
Subsection (loops and conditionals)	Submethod
In-line (specific line of code)	In-line
Variable definition	Variable definition

Table 2: Our modification of the Soudian-Fels system of structural categorization. The categories are listed in increasing specificity from top to bottom.

Soudian & Fels tried to categorize programmer comments in a similar way in their user studies with the Verbal Source Code Descriptor [28]. They categorized users' transcribed comments into the groups shown in Table 2, based on the location users placed the transcriptions of their voice comments. They assumed that the locations the users placed the transcriptions reflected the structures the programmers were focused on when generating a given comment. Although Soudian & Fels referred to the categories they assigned as "structural," what they actually tried to describe was the attentional state of the programmers.

Soudian & Fels's categorization scheme was created for user studies of the Verbal Source Code Descriptor, which attached transcripts of spoken comments to particular locations in the code. In our recording studies, however, programmers were not instructed to place their comments anywhere in the code. The subjects' highlighted region histories indicate that they often made comments when the cursor was between method boundaries, an editor state which causes Eclipse to highlight the entire class.

Instead of basing our attentional classification on location, we assign attentional categories based on referring expressions in the comments themselves. According to Grosz & Sidner, the attentional state is often explicitly mentioned in the comment unit with which it is associated. For example, if the subject of a comment unit refers to a method, then the comment is labelled a method-level comment. The subject is defined topically rather than grammatically: it is not always the subject of an utterance; instead, it is usually the target of some commentary on functionality, purpose, or change in value. If the comment refers to a loop, as in "now we cycle through all the nodes," then the comment is labelled a submethod comment. Because subjects often refer to identifiers in their code, classifying the comments is usually straightforward for a human annotator. Table 2 shows our adjustments of the Soudian-Fels categories to match Java's grammar and the experimental programming task.

Sometimes subjects attend to and mention code entities at different levels of the structural hierarchy in the same comment unit. For example, subjects may talk about the contribution of a variable to a function's processing. For consistency, we classified any cross-structural comments as the more specific of the possible hierarchical categories. For example, a comment on both a variable and a method would be classified as a variable comment.

Subject	Total	Class	Method	Sub-section	Variable	Line	Overall
1	37	2	14	12	4	3	2
2	44	5	19	9	4	2	5
3	61	7	23	14	8	4	5
4	35	5	15	5	4	5	1
5	36	7	18	3	2	2	4
Mean	42.6	5.2	17.8	8.6	4.4	3.2	3.4
Subject		Class	Method	Sub-section	Variable	Line	Overall
1		5.41%	37.84%	32.43%	10.81%	8.11%	5.41%
2		11.36%	43.18%	20.45%	9.09%	4.55%	11.36%
3		11.48%	37.70%	22.95%	13.11%	6.56%	8.20%
4		14.29%	42.86%	14.29%	11.43%	14.29%	2.86%
5		19.44%	50.00%	8.33%	5.56%	5.56%	11.11%
Mean		12.21%	41.78%	20.19%	10.33%	7.51%	7.98%

Table 3: Attentional classification of subject comment units.

Only one annotator, the author, labelled the sound files. Although multiple annotators and cross-annotator agreement would bolster our results, this extension to our analysis is reserved for future work. Table 3 shows the results of categorizing subject voice topic segments attentionally, using the method described above.

5.2.2 Intentional Structure

The most important information comments convey is not *what* structures the programmers were talking about, but rather, the *purpose* of the comment with respect to the structures and ideas it refers to. This is what Grosz & Sidner dub the *intention* of a discourse segment. In their theory, the intentional structure gives rise to both the linguistic and attentional structures. Thus, understanding programmers' intentions for making each comment unit is crucial to a software collaborator

that needs to know how to record each comment with the code.

A qualitative review of the recordings indicated that identifying the editing action that corresponded with any given comment unit would not be a difficult task. Subjects often talked about what they were "about to do," "just did" or "will do in a little while," or evaluated changes they had made or were about to make. Thus, we can characterize a majority of the voice comments and their relationship to the code edit history by evaluating each code segment according to its intentional relationship to the code changes to which the comments refer.

The comment unit intentions of the programmers can be divided into two major groups:

1. Intentions *of the code*. Comment units in this group discuss the purpose of program constructs and editing actions.

2. Intentions *of the documentation itself*. Comment units in this group express the speaking programmer's desire that the audience understand something about the code or coding process.

Below, we give detailed descriptions of the intentional categories in each intentional group and list examples of each intentional category from subjects' transcripts. An example is given from both experienced and inexperienced programmers' transcripts to illustrate the differences and similarities between experienced and novice Java programmers'

comments within the same category. Overall, there is no significant difference in the level of detail between the experienced and inexperienced programmers' comments.

- **Intentions to Implement**

  This intentional group contains three hierarchically related intentional categories, each of which is loosely associated with a program structure. In these comments, programmers express *what they intend to make the code do.* The hierarchy of implementation intentions captures the fact that one intended task often comprises several subtasks, some of which have their own intentional comments. In the language of Grosz & Sidner [12], intentions about the overall task are said to *dominate* intentions about the composing subtasks. For example, a conditional inside a method must be implemented for the entire method to return the desired value. Expressed intentions about the method's functionality dominate intentions about what the loops and conditionals inside the method do.

  Three intentional categories were sufficient to capture the dominance relationships between comments that expressed implementation intentions.

  – **Plan**

    Plan comments dominate all other intentional categories and

express how the programmer intends to approach the problem and primarily discuss data structures the programmer plans to use and how they will contribute to the functionality of the API. Plan comments are concentrated in the first five minutes of subjects' recordings, and correspond with the Class focus space. Plan comments occur before their associated editing actions, but they are separated from the coding that implements any of the ideas conveyed in the comment itself. As the sample Plan comments below illustrate, Plan comments often contain information about how a programmer arrived at a particular solution to or representation of a problem.

* Subject 2: "So it seems intuitive to implement nodes as an array of objects, but because this is a multigraph, the edges can't simply be a 2-dimensional array, so it will have to be a vector of objects, with a to field and a from field."
* Subject 5: "So the first thing that occurs to me is that we're gonna need some form of data structure to collect, to represent, nodes in the...so I'm gonna...add...a...collection for, it doesn't have to be sorted...so...let's just do a list type type type."

– **Immediate Plan**

Immediate Plan (IP) comments are dominated by Plan com-

ments and dominate Simultaneous Plan comments (below). IP comments convey intentions that the programmer will implement a small algorithm that contributes to the programmer's overall approach to solving the problem of implementing the class. IP comments often refer to part of a function or, if the function is short, to an entire function.

Immediate Plan comments are named for the fact that they occur right before an edit action, sometimes overlapping the beginning of a long edit. They are usually spoken during a pause in editing or a string of navigational keystrokes. IP comments often but do not necessarily include the preparatory phrase "going to." Generally, no changes occur to the code during the comment, and the comments themselves contain identifiers that refer to code entities.

Examples of IP comments include the following:

* Subject 2: "Clearly, I need to write a function that goes through and searches for elements."
* Subject 5: "Now one thing I'm thinking about in terms of validation here is that when we're adding an edge we need to make sure both nodes exist."

As these examples demonstrate, IP comments may serve as a bridge between the previous action. More importantly, IP com-

ments contain a high-level description of what the programmer is about to do. IP comments are usually tied to a method or substructure of a method, such as a loop that the programmer is about to implement and wants to explain. Sometimes, IP comments describe a series of related actions. For instance, after his statement above, Subject 2 moved through the class adding `public` to the declarations of the inner classes and the signatures of their constituent functions. Subject 5, on the other hand, spoke very specifically about a conditional he was about to begin writing.

– **Simultaneous Plan**

Simultaneous Plan (SP) comments express the intention of the programmer to write a specific variable, loop, or line of code that performs a task which contributes to a dominating IP comment. SP comments are named for the fact that they are most often spoken during typing. Common indicators of SP comments are "right now" and "I am."

* Subject 4: "So I'm writing a for loop to cycle through all the edges and copy them to the new array, the new matrix."
* Subject 5: "Import java dot util dot star, because I'm not sure I'll need an ArrayList or whatever."

Simultaneous Plan comments are similar to Dictation comments

(explained below) in that they contain syntactic information, such as "java dot util dot star" and "return string," but these syntactic elements do not align precisely with the typing as they do in Dictation comments. Simultaneous Plan comments also contain intentional or algorithmic information: Subject 4, for instance, describes the high-level intention of the for loop he is writing, and Subject 5 explains why he is adding a general `import` command rather than the specific `import` command he actually needs.

– **Dictation**

Dictation is defined as reading along with typing and does not generally contain information about the intentions or functionality of code. Programmers literally say names of identifiers or operators *as they type them* or read documentation aloud. Subjects varied greatly in their usage of dictation: Subjects 1, 2, 4, and 5 dictated constantly, but Subject 3 preferred to type in silence. Because of this large variation, and because dictation adds no new information about the functional or intentional meaning or usage of program components, Dictation comments were eliminated from further analysis.

Because Dictation comments, when present, always occurred simultaneously with corresponding text changes, they may even-

tually be useful in a voice annotation tool as a guide for knowing where the programmer's attention is currently focused in the code.

- **Intentions to Instruct**

  Sometimes the programmer speaks directly to the audience with didactic intent. The comments in the two intentional categories below express the speaking programmer's intentions about the audience's understanding of the code that the speaker is writing.

  - **Explanation**

    Explanation comments are made to convey knowledge about the functionality of specific structures in the code to the listener. Instead of evaluating the code on an algorithmic level, Explanation comments occur immediately after their related code edits and refer to and explain constructs in the program.

    Explanation comments are generally dense with identifiers and programming jargon, because the comments review the functionality and purpose of identifiers or program constructs. As with IP comments, during Explanation comments, few to no changes occur in the code. Explanation comments may be identifiable by navigation actions occurring in the editor: programmers may scroll up or cursor back over what they just wrote as

they are explaining it. For instance,

* Subject 3: "So the idea is that if we're in the loop we set edge to remove to be equal to this one, and we check if we didn't find the edge...throw an exception."
* Subject 4: "OK so I coded up the nested for loops that cycle through the original array, copying over the original array unless it corresponds to the node being removed, in which case it skips it, and I'm using the booleans like I explained before."

Subject 4's comment illustrates a problem unique to voice annotation: linking audio that refers to voice comments recorded earlier ("like I explained before") or that have not yet been recorded ("I will explain that later") with the audio they refer to is not as simple as creating a hyperlink in a typed document. Cross-referencing between sound files is a challenging problem for automatic documentation systems for several reasons:

1. Vague references like "explained before" do not contain information about how far back in the comment history the system should search for the referenced comment nor about what earlier comments are relevant candidates for linkage.
2. No person or documentation system can predict exactly what comments will be made in the future and when a reference

to a comment that does not yet exist should be activated and hooked to a specific comment unit.

3. The term "explain" can refer to several comment units. A complete explanation often requires more than one comment. It is difficult to determine when all of the relevant comments occurred or will occur, especially if they were not or will not be made in one contiguous stretch of time.

The notion of audio hyperlinking has not appeared in voice annotation literature and provides numerous avenues for future exploration. An interface needs to be developed to allow programmers to identify audio recordings they refer to when they leave a piece of documentation that cross references other voice documentation. Programmers listening to or reading the transcripts of voice annotations also need a way to browse the links between voice annotations.

– **Reflection**

Reflection comments express the programmer's intention that his or her listeners be able to evaluate the code critically. Reflection comments frequently discuss what could have been done better, what is suboptimal, and what assumptions the programmer has made and what would need to be changed if the assumptions are not true. The past tense is frequently but not

necessarily used in Reflection comments, and Reflection comments refer to groups of editing actions made earlier in the coding session.

* Subject 2: "So the advantage of using a vector is that I can go through the nodes before, but the disadvantage, certainly, is that I'm going to go through all the nodes that I've gone through before, but of course for each node hopefully I won't have to do it too many times...so it's n cubed time, but it's a reachability algorithm."
* Subject 5: "Now an ArrayList is a bit of a problem because two nodes could be equal but have different object references, so I'm not sure."

As the above examples demonstrate, Reflection comments are as often about what was *not* done as what was. Subject 5, for example, talks about *not* accounting for "different object references" in his node equality testing. Programmers who must use Subject 5's code in the future will need this information. However, rarely are these passing observations typed into the program listing: Reflections are quicker to speak than to type into the code because often these observations occur to speakers when they have already moved their cursor far from the appropriate commenting location and must spend effort losing their

current place in the code and train of thought.

– **External**

External comments express intentions that the audience understand something about the coding process, rather than about the code's algorithms and functionality. Sometimes subjects commented on the quality of the programming environment or expressed their opinions of the Java language. An example from Subject 4's transcript is, "If you don't mind, I dont remember any of Java, so I'm just guessing at some of the syntax." These comments occur rarely and do not correspond to any editing actions.

– **Boundary/Task-Switch**

Boundary/Task-Switch (BTS) comments express the programmer's intention that the audience know the speaker is moving to a new task or is completing the one he or she has been working on. BTS comments are brief exclamations that occur after string of edits, identifiable usually as the word "that's" plus a brief evaluative term, like "done" or "good." Two examples are,

* Subject 2: "And that is my reachability function."
* Subject 5: "OK, I think the API's complete."

Subject	Total	Ext	Exp	BTS	P	Ref	Sim	IP
1	37	3	1	2	3	2	6	20
2	44	1	0	3	8	7	2	23
3	61	7	8	0	3	5	8	30
4	35	2	7	1	2	2	3	18
5	36	2	2	3	3	4	5	17
Mean	42.6	3	3.6	1.8	3.8	4	4.8	21.6
Subject		Ext	Exp	BTS	P	Ref	Sim	IP
1		8.11%	2.70%	5.41%	8.11%	5.41%	16.22%	54.05%
2		2.27%	0.00%	6.82%	18.18%	15.91%	4.55%	52.27%
3		11.48%	13.11%	0.00%	4.92%	8.20%	13.11%	49.18%
4		5.71%	20.00%	2.86%	5.71%	5.71%	8.57%	51.43%
5		5.56%	5.56%	8.33%	8.33%	11.11%	13.89%	47.22%
Mean		6.63%	8.27%	4.68%	9.05%	9.27%	11.27%	50.83%

Table 4: Intentional labels of subject comments. Abbreviations are as follows: Ext - External, Exp - Explanation, BTS - Boundary/Task-Switch, P - Plan, Ref - Reflection, Sim - Simultaneous, IP - Immediate Plan.

Comments following BTS comments are usually IP or Plan comments that start the next implementation task.

Table 4 shows the distribution of comment units over intentional categories.

5.2.3 Summary of Comment Unit Categories

In summary, we labelled programmers' comment units along two dimensions which correspond to two components of discourse structure from Grosz & Sidner's theory of discourse [12]. The first relates to attentional information and tries to identify attentional structure of the comments from the identifiers they contain. The second describes intentional structure and identifies the purpose of the programmers' comments with respect to the audience, the code, and the speaking programmers themselves.

6 Discussion

6.1 Linguistic Segmentation

Segmentation of the audio files into comment units was difficult because users pause irregularly in the middle of phrases when they are distracted by coding or stop to think about the problem before finishing or changing their comment. The five second rule was not strictly followed based on the annotator's discretion about the beginnings and ends of topics in the subjects' commentaries. Sometimes pauses slightly longer than five seconds were ignored if the speaking programmer resumed his or her sentence after a long pause. As the total counts of comment units for the subjects in Tables 3 and 4 show, some subjects like to comment significantly more than others. The total number of comments across an hour varied from 35 to 61 comments: some subjects made twice as many distinct comments as others.

Comment unit segmentation presents challenges for an automatic documentation system that are not problems for a human annotator experienced in programming. Sound processing can only detect pitch, pauses, and other acoustic changes. Many of the boundaries between comments, however, were placed based on changes in *topic*. Techniques from work on topic segmentation may be adapted to the voice annotation domain and applied real-time to enable automatic documentation

systems to detect topic boundaries in audio input [14, 2].

6.2 Verbal Comment Categorization

6.2.1 Attentional Classification

The results from this study affirm Soudian & Fels's finding that most comments are about functions and that the second most popular attentional category is the submethod construct[28]. This trend reflects the programming task: because subjects are asked to fill in the bodies of methods, they attend to and talk about methods and their major components. As discourse theory predicts, the structure of the programming task affects the structure of the programmers' discourse [12].

Attentional categories based on programming language constructs were sometimes difficult to assign. Subjects' comments often wandered between categories unsystematically, and parts of comments about classes would be about the data members within them. For example, Subject 4 stated, "And that can be my toString function, and at the end I'm just going to return string, the one that I've created." This comment unit mentions a method ("toString"), a variable ("string"), and an assignment operation ("created"). Like many evaluative or explanatory comments, this comment does not belong exclusively to one Soudian-Fels category. Indeed, many valuable comments that document decisions made in the API implementation affect and mention all program

structures. As noted in Section 4.2.1, we uniformly labelled mixed comments with the most fine-grained program structure referred to in the comment. The point of attentional categorization with respect to program structures was to determine how much programmers would discuss and document each type of structure in the code. In many cases, it was impossible to assign one attentional state to a particularly long or complex comment.

6.2.2 Intentional Classification

The results for this classification dimension are striking: most comments made were Immediate Plan comments, whereas the number of comments in other intentional categories vary significantly from subject to subject. All subjects, no matter how much they differed in other temporal categories, used roughly half of their speaking phrases to tell the user and to spell out for themselves what they are "going to do" in the very near future.

In roughly one third of the IP comments, subjects start speaking but begin coding very shortly before they finish speaking. It is likely that subjects do not want to forget the algorithm or change that they wanted to implement in the course of their comment, and have some limited capacity for simultaneously typing code and speaking about it. It is uncertain how limited this capacity is or how it varies across different

programmers. A similar phenomenon occurs in pair programming: As Williams *et al.* note, when one partner types, the other thinks at a more strategic level [32]. When there is only one programmer, that programmer is responsible for both typing and strategizing, and when the programmer talks during coding, he or she speaks primarily at a syntactic rather than strategic level.

Taken together, these results suggest that programmers may not comment so much textually because they do their thinking and planning beforehand in their minds, sometimes discussing their plans with themselves or a partner in the room, and do no want to go back afterwards because they are moving on to the next subtask. According to one programmer on the Fog Creek Software Forum [15],

> I think most programmers have an inherent dislike for staying on the same problem after its been fixed (i.e., commenting the code they just wrote – its usually incredibly obvious at that moment what the code does, so why comment?).
>
> I find that I usually comment at my best when I comment before or during coding, since at that stage I'm still figuring out the solution etc. If I wait until afterwards, my results tend to be slapdash and haphazard.

Our results indicate that this programmer is not alone in his preference for commenting and documenting as his implementation ideas first occur to him rather than going back through his code afterward. Subjects seldom went back over their code to explain it. In fact, the subject

with the highest percentage of Explanation comments said them all in one four-minute stretch at the end of his recording, after he had finished his class. This phenomenon parallels the behavior of programmers who do not comment at all while coding and leave all documentation till the end, when they may have forgotten much of their code. Instead, subjects use many IP comments, which serve as both a self-affirmation of what they are planning to implement in the very near future (instructions for themselves), and as documentation for the potential future listener (a record of their intentions). This tendency indicates that a voice annotation tool *can* be useful to programmers because voice annotations capture programmers' thoughts, allow programmers to check their own plans for implementation by reciting the plans to themselves before actually typing them out, and save documentation for the code's users and inheritors. Furthermore, the predictability of the timing of intentional categories suggests that a simple algorithm may be devised to predict and locate relevant intentional information based on the timestamp of edit history boundaries.

The subjects' annotation timelines (Appendix H) show the predictable time relationships of each intentional category with respect to the coding actions they refer to. Plan comments are largely stated before all other comments. Sometimes, as with Subject 1, Plan comments appear in the middle of the programming session, when the programmer

switches to and strategizes about his next few programming moves. IP comments are sometimes separated by SP comments, and these SP comments are subtasks of the IP intentions. Explanation and Reflection comments, because they are intended to help the audience understand code that already exists, both occur after their relevant coding actions. BTS phrases are spread throughout the coding sessions.

6.3 Improved Documentation with Voice Annotation

Our hypothesis proved true: a voice annotation tool can encourage programmers to leave better documentation of their code. Although subjects were given the same instructions to document their code as they are usually given in coursework or in the workplace, they left much richer comments than are usually found in textual documentation. Moreover, subjects commented constantly throughout the programming session.

When asked whether and how their programming speech behavior changed with a voice annotation recorder, subjects testified that they rarely speak to themselves while programming and were uncertain when and how much they should speak. When subjects were not prompted to record their voices, we found that programmers rarely speak about their code beyond a few exclamations. Whether programmers have a voice recorder or not, the textual documentation they leave does not

contain the same deliberation or detail that their verbal documentation does.

There are several reasons that textual comments do not contain as much information as voice annotations do. The most obvious is sheer quantity: As Soudian & Fels showed in their user study with the Verbal Source Code Descriptor, programmers comment less when they type than when they speak their comments, in terms of the number of words used [28]. It is easier for programmers to say things than to type them, especially if their hands are already engaged in coding [3]. In fact, Soudian & Fels found that programmers left 50% as many textual comments when they had to type as when they could dictate their comments using the VSCD.

More importantly, however, programmers leave qualitatively and functionally different information in voice annotations than in text annotations. Because text comments are written *after* the code they comment has already been completed, programmers usually only record what the code does. The standard function header describes the inputs, outputs, return values, and side effects, and, occasionally, if the programmer is feeling particularly energetic or ambitious, special conditions under which the function may (or must) be called. Often, as programmers themselves have noticed, only the API is documented and the algorithmic constructs within the functions are left unexplained

[15].

On the other hand, voice annotations, as we have demonstrated, are most frequently created *before* their corresponding editing actions take place. Thus, a voice annotation tool can record programmers' design decisions as they are made. For instance, a significant percentage of comment units were Plan comments made about the overall approach and Reflection comments made about tradeoffs taken for particular representational decisions. Such deliberations were easily captured in sound but are rarely seen in file or function headers. Furthermore, programmers leave instructions for themselves or the inheritors of their code. In Explanation and Reflection comments, remarks like "Note that I am assuming" and "if you need to change this" appeared in all subjects' transcripts. Such instructions may prove valuable to future attempts to change and extend the code written.

In addition to capturing intentions, vocal expression can also help the programmer check syntax, clarify the programming task, and think through their algorithms before adding them to a program. For example, subjects left instructions on how to change the code for their anticipated future listeners. The enunciation of such instructions both documents design issues that future implementors must take into consideration and allows the single programmers to examine their own design decisions in light of what the inheritors of their code may need

to do. For example, Subject 4 changed the internal representation of the multigraph's nodes to inner classes of the Graph class when he realized future programmers may want to differentiate nodes by their names:

> Now keep in mind that I'm assuming that nodes, when they're passed into these functions, map easily to an integer position in the matrix, like their row number, but obviously, they wouldn't be that simple, you need some sort of mapping function for a node object that contains a...contains a string...perhaps we could create a list of strings, of the nodes we've seen, and have associated with each node it's row number columns number but, you know...what I'll do is, I'll assume nodes are objects that contain a string field, so I'll have an ArrayList of stringsor actually, I'll create a new class, which I will call GraphNode, and that class will contain the node name, which I'll label it, and it will contain an integer which corresponds to the row the node is in, in the matrix.

Thus, when Subject 4 considered what the future programmers to whom he was giving instructions would need his code to do, he further modularized his multigraph implementation and took advantage of Java's nested classes feature to create arguably better object-oriented code.

6.3.1 Pair Programming Behavior in Single Programmers

A qualitative review of single programmers' vocalizations reveals that the discourse the VoiceNotes recorder and our task instructions elicited from the subjects contained several elements also seen in pair program-

ming discourse. In a pilot study of pair programmers (Appendix F), we defined four functional categories of the comments partners exchanged based on their contribution to the programming task. All of these functional comment categories also occurred in single programmers when they knew that VoiceNotes was recording their comments. An example from each of the categories is given below:

- **Syntactic verification**

  While pair programmers used conversation to keep checks on their syntax, single programmers dictated their code as they wrote it and read aloud from Java API documentation. Even the subject that did not dictate his code constantly responded to the Eclipse Java editor's notifications about syntax or other errors. Subject 5, for example, asked, "Why is it complaining about ArrayList not compiling?" and then answered his own question: "So...import java dot util dot star because I'm not sure I'll need an ArrayList or whatever." Not only does he notice the error and, through the question, prompt himself to fix it, but he also leaves a voice comment that both documents and motivates the existence of the `import` statement he then creates.

- **Procedural development and confirmation**

  Subject 5 left an excellent example of a voice annotation that both

documents and develops an algorithm to solve a problem in the code:

> Now we need to do reachable...so I suspect this means...can one node object find another node object in any way...so what we need to do is look through all the...so this is gonna be, what this is gonna be is gonna be a recursive problem, is that true, I suppose that is, need to be able to find all the edges that have a from node equal to the from node parameter, and if that matches the toNode we have in our parameter list, then, o happy day, we have our solution, otherwise what we need to do is...call reachable again, with the to node in the original edge as the fromNode, and at the end of the day we find we cannot resolve this then oh boy.

This voice comment provides listeners with a high-level description of the `reachable` method that Subject 5 is about to implement. It is evident that the recursive solution occurred to the subject as he was talking himself through the problem. Although a pair programming partner is not present to work with him to improve on his initial algorithm, the presence of a voice annotation recorder successfully prompted Subject 5 to put his entire plan for implementation into words. Although he may have done the same meticulous thinking silently without a voice recorder present, he may understand his choices more deeply after articulating them to another "person," a presence supplanted by the VoiceNotes recorder.

- **Problem restatement**

In pair programming, while one partner "drives" (codes), the other has his hands and eyes free to reread task specifications and find reference material, if necessary. Single programmers must divert their attention away from their code whenever they want to reread the problem description they have been given. Single programmers seem to avoid this diversion as much as possible: All five subjects read the task sheet multiple times before beginning the recording. During the recording, other than a few glances back at the task sheet, subjects relied on the given API for guidance. For example, Subject 3 asked "So now we want to be able to add nodes and add edges, so why do we do add nodes first" as he read over the task instructions again. In real world software development, a well-formed API with method signatures usually does not exist to guide programmers along their tasks, and programmers must consult requirements frequently. Nevertheless, this behavior indicates an unwillingness to break attention from coding.

- **Intent documentation**

  Once an algorithm has been settled and is beginning to be set in code, single programmers often describe the high-level purpose of each code structure, often a method or a looping or conditional construct, that implements the chosen algorithm, right before they begin typing. For instance, Subject 4 described his Graph construc-

tor before writing any code: "This function is just gonna initialize everything but it won't be doing too much work because it takes no arguments so it's not going to be adding any nodes or edges or anything like that." Such comments seem to help programmers keep track of intermediate subgoals of their programming task by allowing them to enunciate and affirm the purpose of what they are about to write.

6.4 Implications for a Voice Annotations Tool

6.4.1 Time Patterns of Different Intentional Categories

Predictable temporal relationships between the expressed intentions of programmers and the relevant coding actions can help an automatic voice annotation tool track the structure of the task. We found a strong relationship between the type of intention conveyed in a comment unit and its time relationship to its related editing action. Figure 3 shows the temporal patterns of the seven intentional categories. These temporal patterns indicate an overall intentional structure of the programming task and discourse: Plan comments, which express intentions that dominate the whole coding process, tend to occur before all coding and commentary. IP comments, whose intended actions occur immediately after the comments themselves, are separated by SP comments about individual lines of code. Intentions that are not about what the pro-

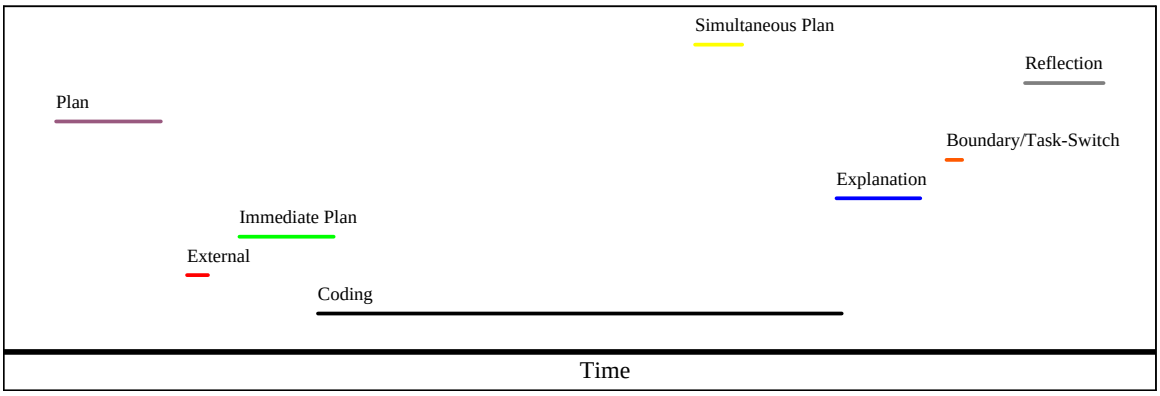

Figure 3: How different intentional categories are related temporally to the relevant coding action.

grammer is doing but rather what the audience should know about the code are expressed after the related coding task is complete, and different intentions about what the audience should believe about the programmer's coding skill, the robustness of the Eclipse editor, and other elements of programming external to the code, are stated at different times.

6.4.2 Task Structure

Subjects tend to group related tasks together. For example, when they define one exception, they immediately define all of the other exceptions. When they define the iterator over the nodes in the graph, they immediately write the iterator over the edges. A voice annotation tool can take advantage of subjects' tendencies to group related coding tasks together by defining hierarchies of tasks to which subjects' verbal comments may be attached. Slaney *et al.*'s work in "cluster[ing] and

segment[ing] sequences of low-level user actions into sequences of distinct high-level user tasks" contains methodologies that may be translated to the programming domain, to cluster and segment sequences of keystrokes into higher-level coding tasks.

If the voice annotation tool can identify editing task segments in the programming session, then even more powerful voice comment processing is possible. For instance, in all recordings but one (Subject 5's), at least one Reflection comment is made about the method just implemented right before the return statement is written. This behavior indicates that the programmer is reviewing his own code and verifying the correctness of his algorithm, and suggests that comments made at or near the end of a method editing sequence can be labelled as comments about the functionality of the method.

6.4.3 Acoustic Markers

Several acoustic characteristics of programmers' voice comments can help a voice annotation tool process their audio input. For example, subjects state comments in different pitches. It may be possible to filter comments below a certain amplitude from a voice annotation system as "comments mumbled to oneself" and retain only higher-amplitude comments meant to be heard by other people. Also, subjects often begin comment units with the word "so." This word may serve as an

indicator of comment unit boundaries. Additionally, the correspondence between dictation and keystrokes can also be exploited to help the voice annotation tool parse the audio stream accurately. Although dictation does not occur in all subjects, a voice annotation system that detects syntax words like "dot" and "star" or programming jargon like "cast" and "loop" can recognize when subjects talk directly about their typing.

Without more robust analytical tools for sound processing and programming task representation, the quantitative validity of these qualitative observations cannot be established. It is very clear, however, even without a quantitative analysis of voice comments, that the voice annotation system captures and encourages the expression of information that is *unnatural* to comment in text but that is valuable to programmers who must work with the code in the future.

6.4.4 Programmer Commenting Style

The large variation of the total number of comment units, the amount of dictation, and comments in the non-IP categories for each subject indicate that prescribing rules for voice commenting behavior across subjects would be not only difficult but uninformative. However, knowing each of these would be helpful for a voice annotation tool, to tell when to record or label an important utterance. Each individual subject's

commenting pattern varied little throughout the hour of programming. Thus, it may be useful to define parameters of voice commenting style that may be automatically or manually quantified, and learned for individual programmers.

Based on this and earlier studies, a programmer's commenting style may be defined by the following parameters:

- the amount of dictation
- the frequency of comments
- the average length of a comment
- the amount of detail each comment contains
- a probability distribution across time categories
- a probability distribution across structural categories

The next step in commenting style analysis is to design, implement, and test the performance of an algorithm for assigning comment phrases to chunks of edit history, whose boundaries are defined by particular keystroke sequences or by different events fired by the Eclipse platform. A model of commenting style based on the above metrics may be devised and trained on a machine learning algorithm to fit different styles of commenting seen in subjects. Such an experiment could also help determine whether commenting styles are continuous over a range of distributions or conform with distinct categories.

6.5 Applications of the VoiceNotes Engine

This thesis only explored VoiceNotes as a documentation tool and introduced the idea of a software collaborator with voice annotation as a component. Two other applications are possible for the engine we have created:

1. **Software development analysis**

   Voice annotation is more likely than existing documentation systems to capture the intentions and design decisions of programmers. Typed comments are necessarily entered before or after the code they document, but voice annotation can capture programmers' thoughts simultaneously as they unfold in code. Analysis of programmers' thought processes, as captured in voice comments, may inform the development of better programming techniques and training for programmers and the improvement of software development procedures for programmer teams in academic institutions and companies.

2. **Search**

   Neuwirth found that spoken comments contain more high-level, colloquial descriptions of the code than do typed documentation [23]. Thus, transcriptions of voice annotations may be useful as a search enhancement for programmers who are unfamiliar with the

code and who do not know the exact identifiers or constructs they are trying to find. A new project member may query the code, for example, with the description, "the part where the expression evaluator checks for malformed expressions." Such a description may well correlate better with "high-level" annotations that would be spoken rather than typed. Although our pair programming pilot study did not involve programmers exploring existing code together, other studies have found honing in on relevant code to be a crucial service collaborators provide each other and thus a valuable feature for a computer collaborator [25].

7 Limitations of This Study

Several experimental limitations may have affected the results of this study:

- **Much information is not currently recorded about the programmer's coding.**

  No voice recognition is used in the VoiceNotes system to transcribe recordings in real time. This design decision was made for three reasons: first, programmers in our user studies and in general coding practice are not instructed or expected to enunciate clearly. Background transcription also slows down the rest of the develop-

ment environment's processes, resulting in delayed editor responses and a poor coding experience. Finally, voice recognition software is poor. For instance, IBM's ViaVoice, one of the best, boasts an average of 96% accuracy in transcribing spoken words to written text [8].

Also, no semantic information is directly recorded in the edit history. This requires a structural parse of the captured text, which would slow down the editor. Nevertheless, semantic information can be inferred from the buffer text dumps in after the programming sessions are done, or recorded directly from program analysis software, such as that provided by the Harmonia Project [4, 13]. To conserve memory and speed, no cursor information is recorded, nor is every keystroke input logged.

- **The granularity of the Eclipse editor's structural tracking may not allow a rigorous analysis of the temporal correlation between edit history and comment units.**

  As noted in Section 3.1, Eclipse highlights the smallest brace-enclosed structure that the cursor is within. Eclipse's algorithm for determining the highlighted structure has several shortcomings. When programmers type within a loop that has not been completed, Eclipse highlights the entire method the loop is a part of,

rather than the local loop region. Also, when programmers move the cursor just outside the right brace of a structure, although they may still be talking and thinking about that structure, Eclipse immediately highlights the next structure up in the program structure hierarchy. Furthermore, Eclipse does not highlight single lines of code, but many comment units refer only to single lines of code.

Thus, although the highlighted region histories provided enough information to allow the annotator to see when edit actions occurred, they did not provide enough granularity to allow the automatic task segmentation of the logged edit histories. More complete information about the code will need to be logged to enable the task segmentation of edit histories and the automatic alignment of task segments with comment units in the audio recordings.

- **The programming task may have been too short or too simple.**

  Subjects were asked to start from skeleton code and choose the internal representation of the `Graph` class. In real-world software development, however, few programmers start programming an API from scratch, and few code for one short hour when they do attack a programming problem. How the time limit on the programming task may have affected subject commenting is unknown, but sub-

jects were well aware of the time pressure: all subjects talked about the time pressure of finishing the "exam." Over a longer period of time or in a more complicated programming task, subjects may adopt different commenting frequencies and styles.

- **Only one professional Java programmer participated in the study.**

  One goal of this study was to find out whether and how experience and programming environment, such as academic versus industry, affects commenting style. Because only one subject was both an experienced programmer and an industry professional, no conclusions about how programming experience and domain affect voice commenting can be firmly drawn from this study.

- **Subjects were unfamiliar with the Eclipse programming environment.**

  Because they had not used the Eclipse JDT regularly in the past, subjects were unaware of the numerous IDE functionalities Eclipse provides. All subjects used Eclipse as a text editor. This likely led to a higher correspondence between speech and typing than would usually be seen. More experienced Eclipse users would likely depend on more IDE-dependent features, such as automatic code generation and import organization, which would be hard to detect

via the simple program text logs used in the VoiceNotes Recorder. Because they spanned a smaller range, subjects' activities were more tractable.

Subjects did not know how to test their code the in Eclipse platform, nor did they understand the various error annotations Eclipse provides to its users. The experienced Java programmer (Subject 5) drew test cases on a piece of scratch paper and talked in detail about each test scenario, but the other subjects spoke little about testing. Because a significant portion of any programmer's time is spent debugging and testing, programmers who know how to perform these tasks with Eclipse should be recorded to characterize voice annotation during testing.

8 Future Work

A number of unanswered questions debated in the initial design and testing of the VoiceNotes system have been raised above. We list below several questions and experiments that provide interesting next steps for this project.

- **Extending the VoiceNotes Platform**

    1. How should associations between audio comments and code, or audio comments and program edit history, be presented to

programmers reviewing or learning about a piece of code?

2. How can we more rigorously and consistently divide voice recordings into comment units? What acoustic patterns may exist in program voice commenting that may aid the automatic segmentation?

3. How can we identify meaningful changes in the code with which comments may be associated?

   The VoiceNotes system implemented and described in this paper includes very naive algorithms for recording edit history. A more extensive programming language analysis engine, such as the Harmonia Project at the University of California Berkeley [4, 13], can be used for semantic analyses of the code that help identify task boundaries in coding. Such boundaries and descriptions of what the programmer is doing in the code can help the automated detection of comment boundaries and provide entities with which audio segments can be associated.

4. Could an algorithm be devised to find comments for each edit history chunk based on the results of this study?

   The results of intentional categorization suggest that phrasing, pausing, and timestamp features of comments may be informative enough to underscore an algorithm for the correlation of task segments in edit history with comment units. Com-

ment unit endings seem to be located on a probability distribution over the timeline surrounding the beginnings of editing segments throughout the edit history. Edit segments may be defined by changes in the highlighted region, as a boundary between a string of navigational cursor movements or a pause in the coding, and a string of non-navigational edits. Information about the time distribution of voice comments can be combined with edit history boundary information to identify the most likely edit actions around which to look for phrases.

- **Exploring Programmer Voice Annotation Behavior**

  1. What should the focus structure for programmer discourse be? Instead of basing attentional categories on programming language structures, focus spaces defined by stages of the programming process may be more intuitive. Possible categories include algorithm design, representation selection, functional decomposition, functional implementation, and debugging. Instead of correlating voice comments with structures in the code, as other systems have tried to do, a voice annotation tool organized around the programming process itself may prove more useful to programmers trying to organize their own thoughts and to programmers trying to understand how a particular im-

plementation was devised.

2. How do programmers comment by voice during later stages of the programming process, such as debugging, building, or testing code?

3. Do programmers program better when they speak than when they do not?

   Because subjects did not compile or test their code in this study, we could neither time how long it took them to complete the API successfully nor could we run their code against test cases to determine the quality of the implementation. Future experiments can compare the quality of voice-commented code against that of text-commented code by timing how long subjects take to complete a class or by running their code against a standard suite of test cases and comparing bug counts across subjects.

Answers to these questions will inform the development of a full voice annotation tool. Once such a tool is developed, we can explore how the long-term usage of voice annotation affect program development and documentation behavior, as well as how different users, such as programmers working on user interface versus on systems, differentially use voice annotation. Programmer voice annotation contains many research possibilities from sound processing, program analysis,

and psychology of programming perspectives.

9 Conclusions

This thesis presented the first step toward a software collaborator for programmers: the VoiceNotes plugin for Eclipse, a voice annotation platform developed to determine whether enabling programmers to document their programs by voice would lead to different and better documentation. We created the VoiceNotes engine to record the voice comments of programmers as they program and to play back recorded programming sessions for annotators to analyze the programming history. We then performed a discourse analysis of programmer comments recorded from a user study with the VoiceNotes tools. We found that an attentional categorization system for programmer voice annotations based on programming language structures needs to be altered to reflect programmers' use of vocalized comments. We also found that there exists a strong programmer preference for describing coding actions immediately before they actually do the coding and that programmers dictate their code as they are typing to varying degrees. Finally, we found that programmers very seldom comment on actions and tasks in the past. We suggested numerous avenues for future studies and proposed a method for correlating voiced comments and edit history segments based on timestamp alignment. Much remains to be

explored in the new domain of voice annotations tools.

10 Acknowledgments

Innumerable thanks go to Professor Barbara Grosz and Professor Stuart Shieber for their invaluable guidance, support, and feedback throughout the formulation, experimentation, and writing of this thesis. I couldn't have asked for better advisors than they.

My thanks also to Ken Shan, Tamara Babaian, and Marco Carbone, who have provided great advice and brainstorms throughout the past year, and to Andy Begel, David Marin, Michael Toomim, and Marat Boshernitsan at UC Berkeley for their help with taming Eclipse, focusing my research, and polishing my prose.

This is the first of I hope many research endeavors, and it has been a valuable one.

References

[1] Babaian, T., Grosz, B., and Shieber, S. 2002. A writer's collaborative aid. Proceedings of the Intelligent User Interfaces Conference. ACM Press.

[2] Beeferman, D., Berger, A., and Lafferty, J. 1999. Statistical models for text segmentation. Machine Learning Special Issue on Natural Language Learning, 34: 177-210.

[3] Andrew Begel. Voice-Over Comments while Programming, http://www.cs.berkeley.edu/~abegel/cs294-1/voice-comments.pdf. CS294 class project. Fall 2002.

[4] Marat Boshernitsan. Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools. Technical Report CSD-01-1149, University of California, Berkeley, June 2001.

[5] Daniel Brantley and David Dillard. Software tools in the service of documentation. In *Third International Conference on Systems Documentation*, pages 6070, 1984.

[6] Satish Chandra Gupta, Tien Nguyen, and Ethan V. Munson, The Software Concordance: A User Interface for Advanced Software Documents, In *Proceedings of 6th IASTED International Conference on Software Engineering and Applications*, November 4-6, MIT, Cambridge, MA, November 2002.

[7] Tzi-cker Chiueh, Wei Wu, and Lap-Chung Lam, Variorum: A Multimedia-Based Program Documentation System, In *IEEE International Conference on Multimedia and Expo (I)*, 2000, pp. 155-158.

[8] CNET Review Article: IBM ViaVoice for Windows Pro USB 10.0. http://att.com.com/IBM_ViaVoice_for_Windows_Pro_USB_10_0/4505-3513_7-20274673-4.html

[9] S. Doerksen, A.P. Shimamura. Source memory enhancement for emotional words. Emotion, 1(1), Mar 2001: 5-11.

[10] *The Eclipse Project*. http://www.eclipse.org.

[11] P.W. Van Gerven, F. Paas, J.J. Van Merrienboer, M. Hendriks, and H.G. Schmidt. The efficiency of multimedia learning into old age. British Journal of Educational Psychology, 73(4), Dec 2003: 489-505.

[12] Grosz, B. and Sidner, C. Attention, intentions, and the structure of discourse. Computational Linguistics, 12(3), 1986: 175-204.

[13] *The Harmonia Project*. http://www.harmonia.cs.berkeley.edu.

[14] Hearst, M. 1997. TextTiling: Segmenting text into multi-paragraph subtopic passages. Computational Linguistics, 23(1): 33-64.

[15] The Joel on Software Forum at Fog Creek Software. "Why Do Programmers Hate Documenting?", http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=showixPost=

[16] A. J. Ko and B. Uttl, Individual Differences in Program Comprehension Strategies in Unfamiliar Environments, In *Proceedings of the IEEE International Workshop on Program Comprehension*, Portland, OR, May 10th-11th, 2003, pp. 175-184.

[17] 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 147153. ACM Press, 1999.

[18] Kraut, R. E., Streeter, Lynn A. (1995). "Coordination in Software Development." In *Communications of the ACM* March 1995(March 1995): 69-81.

[19] Magic Squares. 8th International Olympiad in Informatics, Veszprm, Hungary. http://www.inf.bme.hu/contests/tasks/ioi96d2.html.

[20] Gregory R. McArthur. If writers cant program and programmers cant write, whos writing user documentation? In *Proceedings of the Fourth International Conference on Systems documentation*, pages 6270. ACM Press, 1985.

[21] Nagappan, N., Williams, L., Wiebe, E., Miller, C., Balik, S., Ferzli, M., Petlick, M., Pair Learning: With an Eye Toward Future Success, Extreme Programming/Agile Universe 2003.

[22] *The Natural Programming Project.* http://www-2.cs.cmu.edu/~NatProg/langeval.html.

[23] Neuwirth, C. M., Chandhok, R., Charney, D., Wojahn, P., and Kim, L.. Distributed Collaborative Writing: A Comparison of Spoken and Written Modalities for Reviewing and Revising Documents. In *Conference Proceedings on Human Factors in Computing Systems: CHI 94*, pp. 5157.

[24] Ortiz, Charles L. and Barbara Grosz. 2002. "Interpreting Information Requests in Context: A Collaborative Web Interface for Distant Learning". *Autonomous Agents and Multi-Agent Systems*, 5, 429-465, 2002. Kluwer Academic Publishers.

[25] *The Pair Programming Initiative.* http://www.pairprogramming.com.

[26] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST*, 1997.

[27] Malcolm Slaney and Jayashree Subrahmonia and Paul Maglio, Modeling Multitasking Users In *Spring-Verlag Lecture Notes in Artificial Intelligence*, UM2003, User Modeling: Proceedings of the Ninth International Conference, 22-26 Jun 2003. University of Pittsburgh, Johnstown, USA.

[28] S. f. Soudian and D. L. Fels, Verbal Source Code Descriptor, Prepared for IEEE WESS 2002.

[29] L.J. Stifelman. The Audio Notebook: Paper and Pen Interaction with Structured Speech. Ph.D. dissertation. Massachusetts Institute of Technology. September, 1997.

[30] Stotts, D., Williams, L., Nagappan, N., Baheti, P., Jen, D., Jackson, A., Virtual Teaming: Experiments and Experiences with Distributed Pair Programming, Extreme Programming/Agile Universe 2003.

[31] Laurie Williams, Robert R. Kessler, Ward Cunnngham, and Ron Jeffries. http://collaboration.csc.ncsu.edu/laurie/Papers/.

[32] Williams, Laurie and Kessler, Robert R., "All I Really Need to Know about Pair Programming I Learned In Kindergarten," In *Communications of the ACM*, May 2000.

[33] Williams, Laurie and Upchurch, Richard. "In Support of Student Pair Programming," *2001 SIGCSE Conference on Computer Science Education*, Charlotte, NC, February 2001.

[34] Williams, L., Wiebe, E., Yang, K., Ferzli, M., Miller, C., "In Support of Pair Programming in the Introductory Computer Science Course", *Computer Science Education*, September 2002.

[35] “Pair Programming.” http://c2.com/cgi/wiki?PairProgramming/

Appendix

A Magic Squares

This is the text of the Magic Squares problem, taken from the 8th International Olympiad in Informatics, held in Veszprm, Hungary, in 1996 [19]. We gave this problem to the subjects in our pair programming pilot study, described in detail in Appendix F.

```
Magic Squares Following the success of the magic cube, Mr. Rubik
invented its planar version, called magic squares. This is a sheet
composed of 8 equal-sized squares (see Figure 3).

-----------------------------------------------------------------
1 2 3 4 8 7 6 5

Figure 3: Initial configuration
-----------------------------------------------------------------

In this task we consider the version where each square has a
different colour. Colours are denoted by the first 8 positive
integers (see Figure 3). A sheet configuration is given by the
sequence of colours obtained by reading the colours of the squares
starting at the upper left corner and going in clockwise
direction. For instance, the configuration of Figure 3 is given by
the sequence (1,2,3,4,5,6,7,8). This configuration is the initial
configuration.

Three basic transformations, identified by the letters 'A', 'B'
and 'C', can be applied to a sheet:

'A': exchange the top and bottom row, 'B': single right circular
shifting of the rectangle, 'C': single clockwise rotation of the
middle four squares. All configurations are available using the
three basic transformations.

-----------------------------------------------------------------
```

A: 1 2 3 4 8 7 6 5 1 2 3 4 8 7 6 5

B: 1 2 3 4 4 1 2 3 5 8 7 6 8 7 6 5

C: 1 2 3 4 1 7 2 4 8 6 3 5 8 7 6 5

Figure 4: Basic transformations

The effects of the basic transformations are described in Figure 4. Numbers outside the squares denote square positions. If a square in position p contains number i, it means that after applying the transformation, the square whose position was i before the transformation moves to position p. You are to write a program that computes a sequence of basic transformations that transforms the initial configuration of Figure 3 to a specific target configuration (Subtask A). Two extra points will be given for the solution if the length of the transformation sequence does not exceed 300 (Subtask B).

Input Data The file INPUT.TXT contains 8 positive integers in the first line, the description of the target configuration.

Output Data On the first line of file OUTPUT.TXT your program must write the length L of the transformation sequence. On the following L lines it must write the sequence of identifiers of basic transformations, one letter in the first position of each line.

Tool MTOOL.EXE is a program in the task directory that lets you play with the magic squares. By executing "mtool input.txt output.txt" you can experiment with the target configuration and the sequence of transformations.

Example Input and Output

```
------------------------------------------------------------------
INPUT.TXT                 OUTPUT.TXT

2 6 8 4 5 7 3 1           7
                          B
                          C
                          A
                          B
                          C
                          C
                          B
```

Figure 5: Example Input and Output

B Subject Questionnaire

The following questionnaire was given to subjects before the programming task.

Subject Number ____

Please fill out the following as accurately and precisely as you can:

1. Today's date:
2. Rate your level of programming experience from 1 (very little or no experience) to 10 (very experienced):
3. How many years have you programmed?
4. In the following list, please circle the programming languages that you know. On the left of each language you have circled, write the number that best corresponds with your experience with the language: 0 - use for my own side projects, 1 - used in class on homework assignments, 2 - used in class on a large system, 3 - used in a job or internship. Write all numbers that apply.
   - C, C++

- Java
- Lisp, ML or another functional language
- Prolog
- Perl or another scripting language
- SQL

5. Put a check next to the language you use most frequently.
6. Rate your level of experience with Java from 1 (very little or no experience) to 10 (very experienced):
7. How many years have you programmed in Java?
8. In the following list, please circle the programming environments that you have used. On the left of each item you have circled, write the number that best corresponds with your experience with the environment: 0 - use for my own side projects, 1 - used in class on homework assignments, 2 - used in class on a large system, 3 - used in a job or internship. Write all numbers that apply.

   - Visual Studio
   - Eclipse
   - Emacs/XEmacs
   - `vim` or another text editor

I certify that the above is correct.

Signature of subject:

Date:

C Programmer Testimonials

The Joel on Software Forum at Fog Creek Software conducted a survey on "Why Do Programmers Hate Documenting?" The excerpts below contain some elucidating, if not amusing, insights [15]:

- "I think most programmers have an inherent dislike for staying on the same problem after its been fixed (i.e., commenting the code they just wrote – its usually incredibly obvious at that moment what the code does, so why comment?).

  "I find that I usually comment at my best when I comment before or during coding, since at that stage I'm still figuring out the solution etc. If I wait until afterwards, my results tend to be slapdash and haphazard.

  "Thoughts?"

- "When I am writing code, it all is blazingly obvious to me right now. In the future, someone is going to find some part of it confusing. The problem is that I can't predict /which/ bit is going to be confusing. So which bit do I comment? If I comment every last line, then the comments get tedious, as mentioned in other threads."

- "Programming is a largely a creative, problem-solving effort. Documenting is largely a teaching and communication effort. We like programming because we like creating solutions to problems. (Some of us like finding the solution so much that once we've figured out the solution we're not interested in implementing it!) We don't like documenting because we feel like we're not really solving a problem, we're just describing something.

  "Also, I think a lot of programmers view documentation as largely being a waste of time and effort. When you are writing the equivalent of, To create a new document click 'File', then 'New', the thought going through your head is what a colossal waste of your time this is. After all, anyone can figure out how to do simple things, right?

  "I don't believe the claim that programmers are lazy. I do believe that programmers dislike doing things that are not programming. Like writing documentation, working on schedules, defect tracking. And we'll procrastinate on those things, or do the minimal effort necessary. That's an ego thing."

- "In particular, comments should be used to inform the reader of things like pre- and post-conditions for methods, the intended use

of variables, and anything else which might be non-obvious to someone other than the author (including the author a year from now)....

"Of course you shouldn't comment every line of code. In fact, for most methods you only need a single comment describing the intended use and parameters. But any real-world system will have plenty of cases where explanatory comments are helpful and greatly reduce the time spent maintaining the code, especially by reducing bugs introduced by the maintainers through misunderstanding."

- "Writing source comments isn't bad. The trouble is updating them when the code changes. Source can change a lot in the first cut, and no comments are better than misleading ones.

  "I have just found this out *again* on my current project. Had to change the operation of a server to fix some bugs, and well , any original comments are just not good.

  "Actual real documents are a complete pain. Ditto the updating part. No one reads them either. Developers are frequently under so much time pressure that they have to optimize, and documents that don't get read are the first to go."

- "I don't mind writing some small document to go with my code if its needed. Usually if you do document an API half a page of text per function is plenty. However most places I have seen use totally bloated templates for documentation purposes. These templates usually are one-size-fits-all, used for all program documentation needs regardless of platform, which renders 70% of the template not applicable because it deals with say mainframe related stuff, and I'm documenting UNIX stuff.

  "Short and sweet goes for code. Short and sweet should go for documentation as well. Templates are so stupid it is laughable.

  "This is why I hate writing docs. And needless to say, why people hate reading it."

D Programming Task Instructions

You will have one hour to work on the implementation of the following Java class, `Graph`, whose API is given below. `Graph` represents

a directed, acyclic multi-graph. Each edge in the graph has unique combination of from node, to node, and edge label. Edges between different pairs of nodes may have the same label, and there may be multiple edges between the same two nodes.

Specifically, you will

1. define the internal representation (fields) of the class,
2. implement all of the methods whose signatures are shown, and
3. add methods for at least two of the following functionalities, whose signatures you must define yourself
   - determine the shortest-path between two nodes
   - find a Hamiltonian cycle
   - find a Eulerian cycle
   - find a clique
   - construct the complement of the graph
   - create an iterator over the edges
   - create an iterator over the nodes
   - find the intersection between this graph and another graph

When you are ready to begin the task, press the record button on the VoiceNotes View panel. A timer will appear in the panel to tell you how much time has elapsed. When one hour has passed, please come to a stopping point and press stop when you have finished all edits to the code.

You may use the internet as much as you'd like to look up Java utility definitions.

Please document the code as you are programming. Although you can record your comments with "/* */" and "//" marks in the code, you may find it easier to and are encouraged to speak your comments aloud. At the end of the task, you will be paid between $10 and $20, based on a computerized rating of the quality of verbal comments you leave.

E Graph Class API

This is the Java API subjects were asked to implement and extend in our single programming user study.

```
public class Graph {

    public Graph() {
    }

    void addEdge(String edgeLabel, Object toNode, Object fromNode)
    throws DuplicateException
    {
    }

    void addNode(Object aNode) throws DuplicateException
    {
    }

    void removeEdge(
        String edgeLabel,
        Object toNode,
        Object fromNode)
    {
    }

    void removeNode (Object aNode) throws NotFoundException
    {
    }

    Collection getEdges()
    {
    }

    Collection getNodes()
    {
```

```
    }

    String toString()
    {
    }

    boolean reachable (Object fromNode, Object toNode) {
    }
}
```

F Pair Programming Pilot Study

F.1 Background and Motivation

Pair programming is a rising trend in academia and industry: programmers who write their programs using the pair and extreme programming methodologies report improvements in efficiency, bug-tracking, and overall programming experience. In recent years, several studies have shown that pair programming results in better performance [25]. In one study at the University of Utah, for example, programs created by pair programmers were found to pass, averaged across four programming assignments, 15.2% more test cases than their single-programmed counterparts, and were completed in 40-50% less time [31]. Other studies have shown that pair programmer are more satisfied with their job, learn quickly and tacitly from their partners, and work better in distributed development situations, where most pairs are communicating with others over a distance [18, 21, **?**, 32, 33, 34].

It is possible that the phenomena which occur in pair programming but not in single programming can be encouraged or modelled by machines so that single programmers may get the some of the benefits of pair programming without having a partner present. An exploration of the ways in which pair programming helps partners is required to identify the roots of pair programming's benefits.

Williams & Kessler have conducted several qualitative studies to determine the ways pair programming helps programmers [32]. In their experiments, one element of pair programming stood out as the most

salient, obvious, and crucial difference between pair and single programming: the conversational interchange between partners. That is, division of labor does not account for all of the benefits of pair programming. Rather, something inherent in the *talking* itself between partners makes the programmers work faster and better. Pair programmers themselves agree with this hypothesis. For instance, in one study [31], where a class of 20 juniors and seniors at University of North Carolina worked on pair programming assignments throughout a semester and kept online journals of their experiences, the feedback on the importance of conversation was telling:

> When I explained an idea to my partner, I concentrated on what I was saying, and carefully made things clear and logical because I did not want to confuse my partner and I wanted him to understand what I was talking about. It helped me better understand the problem I was addressing. It also helped me discover some mistakes I had made but did not notice before I talked with my partner.

That is, the heightened concentration on the shared task and pressure to know the algorithm or architecture well enough to explain it to another person solidifies concepts in the speaker's mind, if not the listener's. Indeed, when Kraut & Streeter surveyed 750 working programmers on coordination techniques in software development, they found that the communication technique with both the highest use and the highest value was "discussion with peers" [18].

From the standpoint of the listener, there are a related but different set of benefits. As another student testified,

> One problem with single programming is that you can forget what you are doing and easily get wrapped in a few lines of code, losing the big picture. Your partner is able to constantly review what you do, making sure that it is in line with the product design.

That is, because one partner is not mired in the details of creating code structures, he or she can better help keep the team aware of the big picture, the ultimate goals and intentions of the code they are writing.

This helps the pair make and evaluate progress toward solving the problem it intended to solve.

Although communication has been praised, however, no one has studied the discourse between partners closely and formally. In pair programming literature, only one known attempt has been made to date to capture and describe the dialogue between programmers working together. Cunningham & Beck led a Birds of a Feather workshop at OOPSLA 1997 and looked for patterns in the recorded dialog between programmers as they worked together. They found and labelled several common exchanges [35], such as:

- "Let me drive": Every once in a while, the other partner asks to take over the keyboard and type, or "drive," for a while.
- "N plus one" or "N minus one": Partners notice and correct each other's off-by-one-bugs.
- "Inject:into:": Partners suggest variable names to each other.
- "Trust me": Sometimes a partner has an idea for how to do something, but not the words to explain it. This statement indicates that the partner wants to try implementing an unexpressed idea without interruption. That partners have to explicitly indicate when they are coding without discussion indicates that implementation without dialogue is rare.
- "Look where you would write it": Partners often suggest to each other where methods that may have already been implemented may reside.
- "Ask the computer": This statement means, "Don't reason about what will happen if you do X. Do X and see what happens."
- "Blame yourself first": When partners hit a roadblock, one partner suggests that they leave the keyboard and discuss the current implementation in detail without looking at the code. Often when the driver explains the implementation to the other partner afresh, the elusive bug or logical error becomes obvious.

Cunningham & Beck do not describe how often each of these exchanges occur, nor do they hypothesize how these exchanges help programmers program better when they are together versus when they

work alone. A rich of analyses can be drawn from discourse theory, which holds that discourse depends upon the task at hand. No one has studied whether and how this is the case when programmers talk as they program and, if there is a correspondence between the conversation and the programming, what the nature of the correspondence is.

Thus, we conducted a pilot study of pair programmers working on a small-scale programming problem to determine:

- Characteristics of the verbal exchange between pair programmers
- The ways in which partners' vocalizations reflect and influence the program development process
- The types of documentation pair programmers leave in print
- The types of documentation partners do *not* leave but help the programming process (e.g., "TODO" comments that were eventually removed from the program listing or types of comments found exclusively in conversation

F.2 Methods

Two Harvard College computer science majors were given a programming question from the 8th International Olympiad in Informatics, Veszprm, Hungary, called Magic Squares (Appendix A contains the text of the task) [19]. The programmers were given one hour to complete the programming problem, which neither had seen before, in a language of their choosing, on a platform of their choosing. Both sat at the same computer and were given sheets of scrap paper and one pen. During the hour of programming, they were videotaped by a camera placed at an angle in front of them, behind their machine, so that both partners could be seen at all times. Their typing and other interaction with the computer's input devices were recorded using Lotus ScreenCam (trial version).

F.3 Results and Discussion

Despite the experimental suboptimalities discussed above, we gained insight into several aspects of pair programming by studying this brief

pair programming session. We were primarily interested in how the conversational exchanged contributed to the programming task. Thus, after reviewing the video and screen capture, we divided the conversational exchange into topic segments and classified each into four functional categories:

- Syntactic verification
  A: "So we can read into a variable like this, right?"
  B: "Right."
  In low-level decisions such as syntactic detail and variable naming, the pair often fell into a question-answer back-and-forth. This served as confirmation of syntax that, particularly in an editor without syntax highlighting like `pico`, may result in hard-to-find errors during compilation or runtime. Each partner may know different tools in the language they can leverage to complete the task. The sum of their knowledge not only prevents them from constantly consulting language documentation, but also helps both partners learn aspects of programming languages from each other that they did not already know. This benefit of pair programming is especially salient when the pair is programming in a language with numerous libraries, such as Java, or, in this case, a language with extremely flexible syntax, such as Perl.

- Procedural development and confirmation
  A: "And then we read in the file into an array?"
  B: "Right."
  Programmer A spoke as he began to type the code for file opening and redirecting the stream into a variable. In this exchange and others like it, the partners plan the algorithm together, checking each other in the process, playing devil's advocate or suggesting improvements at each step.

- Problem restatement
  A: "How many were we supposed to have again?"
  B: "Twenty-one."
  This exchange illustrates the role of one partner as the "instruction reader" and the other as the "implementor" or, in pair program-

ming literature, the "driver." In pair programming, the person who is not coding often finds references, reminds the pair of the programming session's goals, and keeps the driver on track toward the overall goal. In single programming, the act of voicing comments to the computer may implicitly serve this role.

- Intent documentation
  A: "Let's print this out for debugging purposes"

  Because the driver in the pair had an audience for his verbal comments, he stated the purpose of most segments of code aloud before actually writing them. In this example, the programmer typed the word "DEBUGGING" at the same time as he made the verbal comment. By declaring the intention of the code before typing it, Programmer A both justified the plan of action he was about to carry out and left a record in his partner's memory about the purpose of the extra code, in case they revisit the code later and he (Programmer A) cannot remember its contribution to the program.

When we examined the videotape and screen capture, one trend in the pair's conversational behavior stood out: the programmers would remain mostly silent as they coded, but after each bout of silence during coding, they would engage in a small discussion of technical and algorithmic issues. Most importantly, they would list the assumptions about the functionality of the code they have already written and what they know at that point in the program.

The expression, discussion, and understanding of assumptions is a particularly significant function of the partners' conversational exchange. Determining the range of inputs a function will get at any point is a difficult task for anyone extending on a piece of software, writing a large program, or coming back to a program. Programmers may take a long time to determine what assumptions have been made so that they may adhere to, eliminate, or change the program functionality. Many bugs are generated by programmers who do not fully grasp or remember the assumptions of functions. When partners discuss these assumptions with their partners, the tradeoffs of different paths at any point in the program become clearer.

Thus, not only does verbal interchange serve important functions in the programming process, but it also potentially provides a rich source of documentation. The benefit of conversational exchange is twofold: It enhances the programming process by helping the partners check syntax, develop more robust algorithms, express and remember assumptions, and confirm plans of action. If the comments spoken by the programmers had been recorded at the time they were made, the motivation, intent, tradeoffs, and decisions made in the development of the code would have been saved for future programmers to review. Much information not recorded in the programmers' textual comments contributes invaluably to how well programmers understand a piece of code during its development and when the code is revisited.

Conversation about the assumptions, algorithms, and intentions of a program, a sort of verbal documentation of the coding process, occur naturally when programmers code together because partners serve as each other's audience and give each other feedback. If vocal expression and exchange can be encouraged in single programmers, then programmers coding alone can experience some of the benefits of verbally documenting their programming.

F.4 Limitations of This Pilot

Several factors could have very well affected this pilot's results. First, the programmers were fully aware of the videotaping and screen-recording and had been told that the study was about "how people program together" and may have altered their interchange because they knew they were being watched. Second, the pair had never met and had significantly different levels of experience with programming: one was a systems programmer with several years of research and part-time industry experience, and the other focused on computational theory. Although this "experience gap" often happens in real pair programming and other collaborative situations, it may have resulted in tension in the conversation. Although this tension may have helped the programmers stay focused on their task, as bantering was less likely to occur, whatever benefits of congeniality and familiarity in a pair programming situation were lost.

Third, the ACM problem was designed to be solved over an entire day

and was too difficult to complete in a one-hour programming session. Thus, the pilot may not have captured the stage of programming in which conversational exchange would be about coding—the programmers spent most of their time rereading and restating the problem, instead of writing code. Rereading, restating, and rethinking problems are as much, if not more, of a part of the programming process as actually translating thoughts into a computer language. Due to the one-hour limit, however, we were unable to capture behavior past the initial attempt to understand the problem and to enter some preliminary code. The pair did not have time to enter into a discussion on code architecture, for example, or finer data structure nuances, and may have paused significantly more than they would in the majority of the coding process.

Finally, technical difficulties restricted the programmers to a circuitous way of editing the code. Pressing the ESC key stopped the screen capture. Thus, the programmers had to type on an unfamiliar editor (`pico`). Being less comfortable with one mode of input may lead to reduced output from other modalities and put concentration on lower-level aspects of programming, such as typing the right keys to format the code syntax.

G VoiceNotes System Components

We created the VoiceNotes system as a plugin for the Java-based Eclipse Platform. VoiceNotes extends Eclipse's Java Development Tools to allow programmers to annotate their program by voice [10]. The Eclipse Platform provides an extensive API for system resource and user interface control for programming environment development and code analysis. The VoiceNotes plugin exploits this machinery to control how sound files are stored, retrieved, displayed, and manipulated by users. The Eclipse platform was chosen for its cross-system interoperability and for the robustness and ease of developing large, enterprise-level software with Java. This flexibility will facilitate later extensions of this simple voice annotation provider to a full computer software development collaborator.

The components of the VoiceNotes plugin are:

- **Voice Annotation Controller**

  The VAC is the functional basis of the entire VoiceNotes system, connecting the annotation sound files with the actions on the files. The VAC's job is to maintain one coherent state for the VoiceNotes engine. For example, it houses the central thread of sound input/output so that no two voice annotations are being processed at the same time and no datastreams conflict.

- **Voice Annotation Action Modules**

  Two actions, defined as modules in the Eclipse Platform, are currently available to programmers from the VoiceNotes system: recording and playback. The VAC controls two objects that each corresponds with an action. These action provider objects may be added to menus and toolbars in other parts of the Eclipse Java Development Tools workspace, but are currently accessible only via the Voice Annotation Viewer.

- **Voice Annotation Viewer**

  The Voice Annotation Viewer (also referred to as the VoiceNotes Viewer) is the primary UI for the voice annotation system, a window onto the current state of the VAC. It provides widgets with which programmers trigger VoiceNotes actions and stays synchronized with the current Java editor to display a listing of relevant voice comments with respect to the document in the editor.

- **Voice Annotation Storage**

  VoiceNotes follows the Eclipse schema for storing and retrieving files. Voice annotation files are associated with the current code file and stored in the .metadata/*project*/.voicenotes subdirectory, where *project* is the Java project to which the annotated file belongs. This location in the Eclipse resource hierarchy was chosen for two reasons: a) voice comments are usually made within a project, and b) the Eclipse Platform forces all system resource usage to be associated with a project in the workspace. The voice documentation files are named vnote*id*.wav, where *id* is a monotonically increasing integer used by all projects in a given Eclipse workspace. This universal ID counter allows the system to be easily adjusted in

the future to allow for voice comment sharing across projects. The edit history associated with each voice annotation file is stored in an logfile marked by the voice comment unit's starting timestamp.

We described the two action modules in detail below.

G.1 VoiceNotes Recorder

The Recorder is the key action module of the VoiceNotes system. All analyses depend on how the voice data is stored once a recording is triggered.

Associated with each voice annotation recording is an `Info` object which keeps track of a list of highlighted text histories and associated timestamps, visible code histories and associated timestamps, the beginning and ending timestamp of the voice annotation, and the program listing at the end of the voice annotation. The highlighted text at any time is the region of code representing the smallest region enclosed by braces within which the cursor lies. For example, in Figure 2, the cursor lies within the `findNode` function, causing the Eclipse environment to highlight the entire `findNode` method listing.

The Eclipse developers designed this feature to give a rough estimate of the program structure the programmer is editing at any given time. We will use it as an indicator for the program structure the programmer's attention is on. The visible region is the code visible to the user at any time in the buffer and gives a rough estimate of programmer attention.

When a recording is triggered, the VoiceNotes Engine listens for changes and edits to the document (program) in the Eclipse Java editor. When a change is detected, both the highlighted and visible regions are recorded along with a timestamp into the `Info` object. At the end of the 30 second buffer, the `Info` object for each voice comment is stored as an XML file and named vn*time*.xml, where *time* is the ending timestamp of the comment unit.

G.2 VoiceNotes Playback Tool

The VoiceNotes playback tool was created for the annotator reviewing sound and code recordings by programmers. The tool enables the real-

time analysis of the correspondence, if any, between speech and edit actions.

During playback, all of log files and their associated voice annotation files are loaded into the environment. The annotator can initiate playback by double-clicking on a voice annotation file name or by pressing the play button. Although the annotator was the only person who used the playback mechanism, any programmer interested in the history of the development of the program can review the coding and speaking history. When playback is triggered, the selected sound file is opened and played, and the corresponding highlighted regions are displayed in Eclipse's console viewer in real time.

The annotator can start playback on any voice annotation and pause, stop, or select another voice annotation. The screen automatically refreshes to display the state of the highlighted region of the program buffer as the programmer was seeing it at the moment the recording was made, thus allowing the annotator to watch program development and listen to audio in synchrony.

H Temporal Patterns of Different Intentional Categories for All Subjects

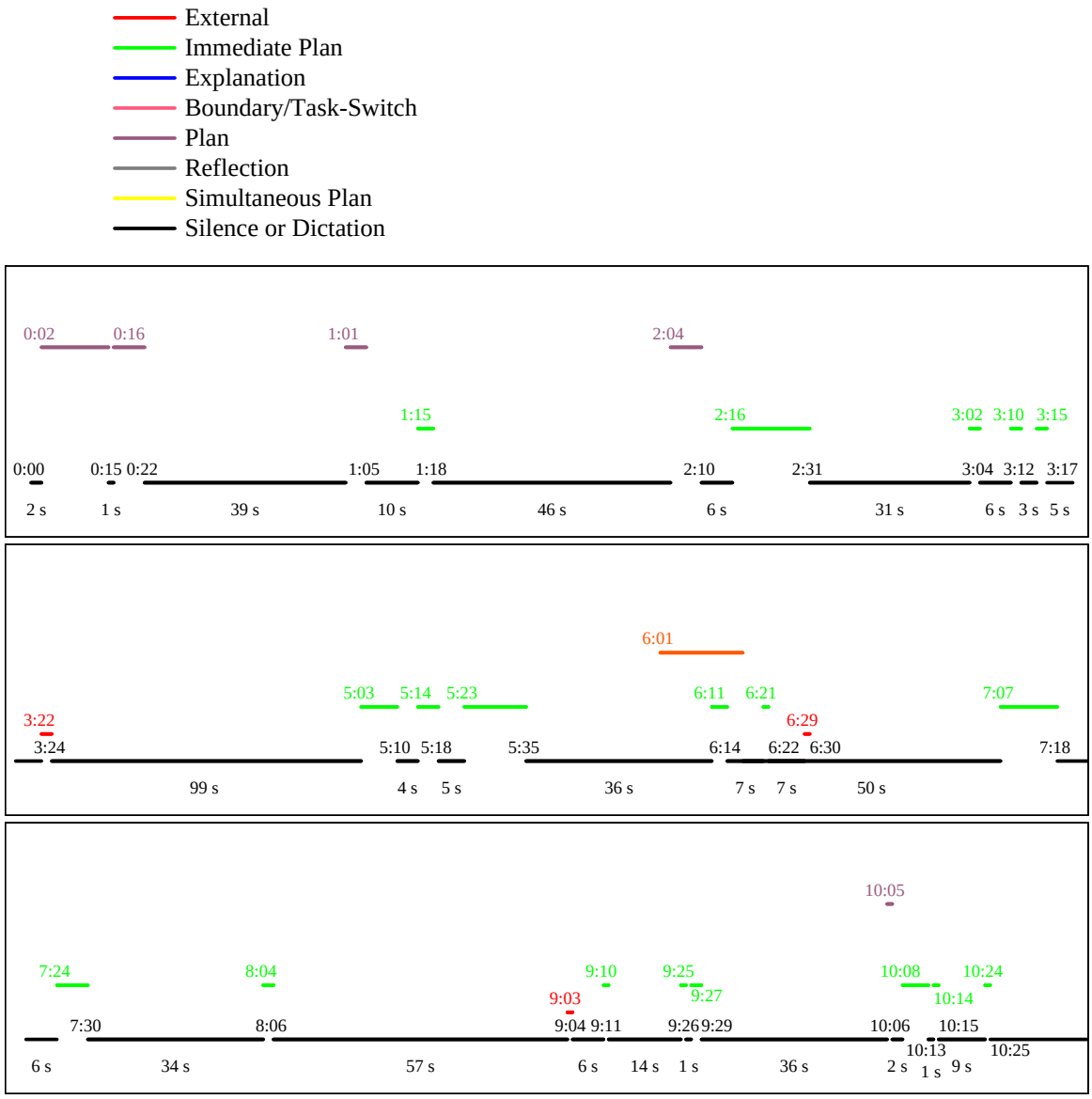

Figure 4: Subject 1's voice annotation timeline.

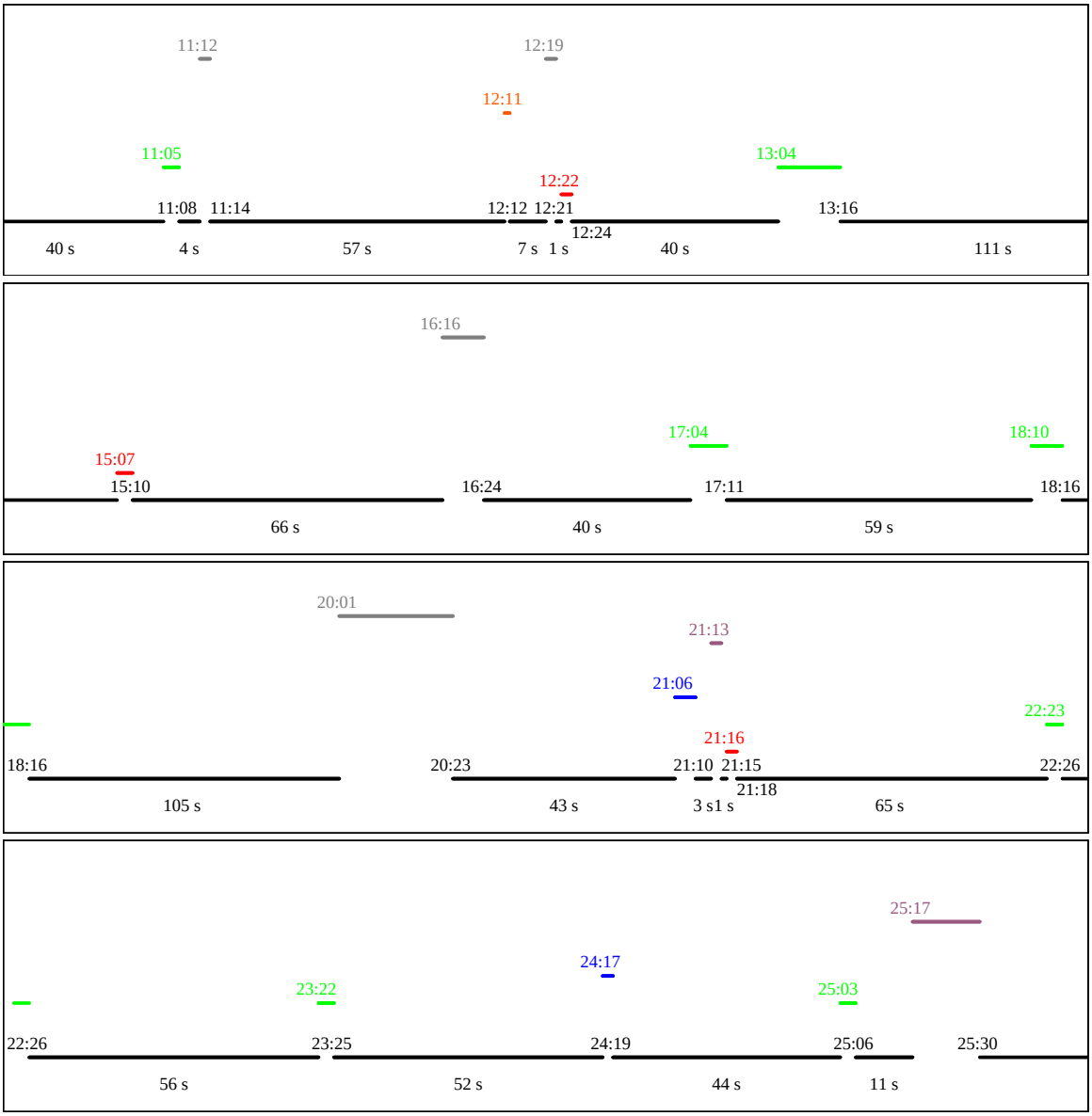

Figure 5: Subject 1's voice annotation timeline, continued.

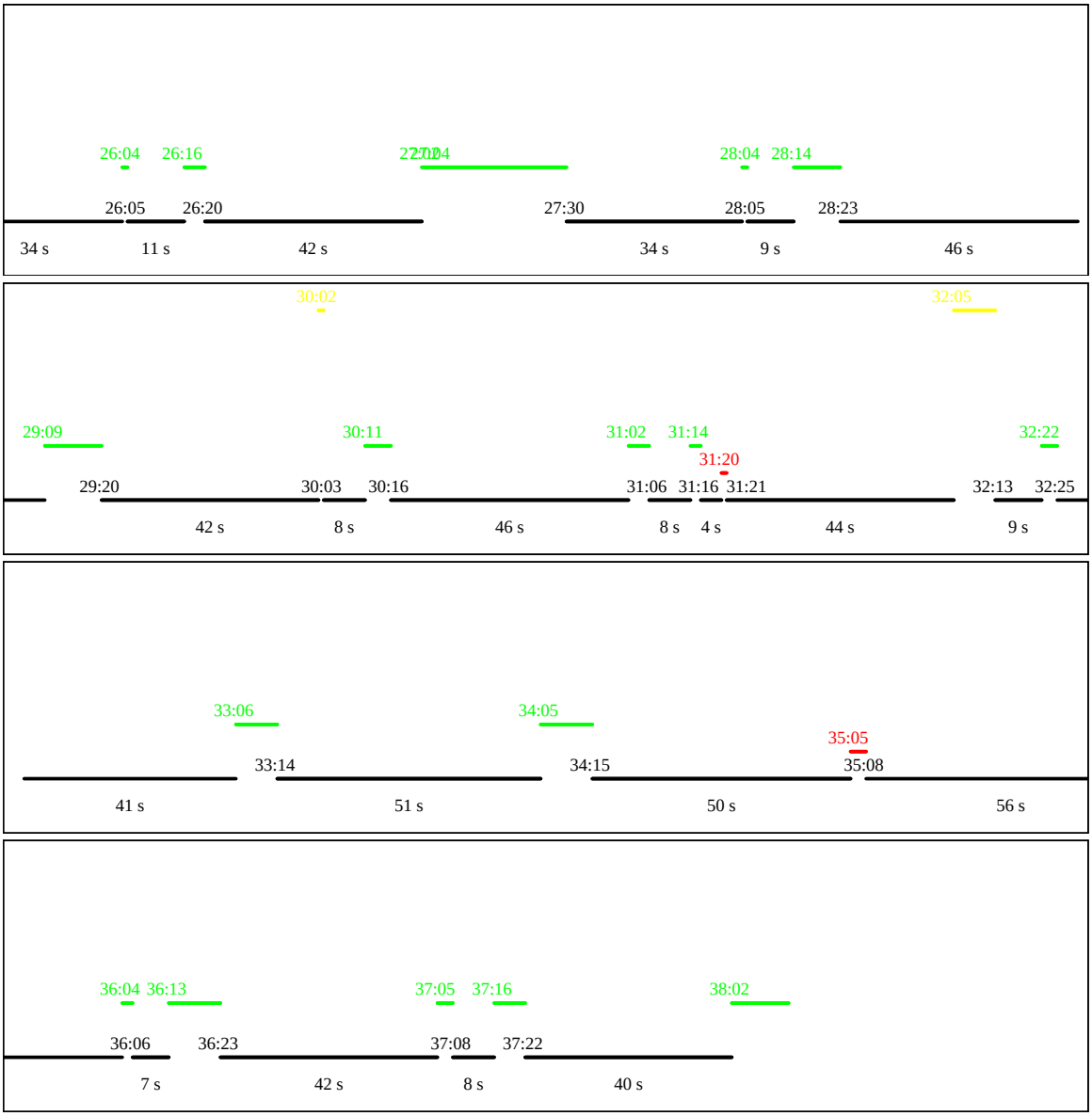

Figure 6: Subject 1's voice annotation timeline, continued.

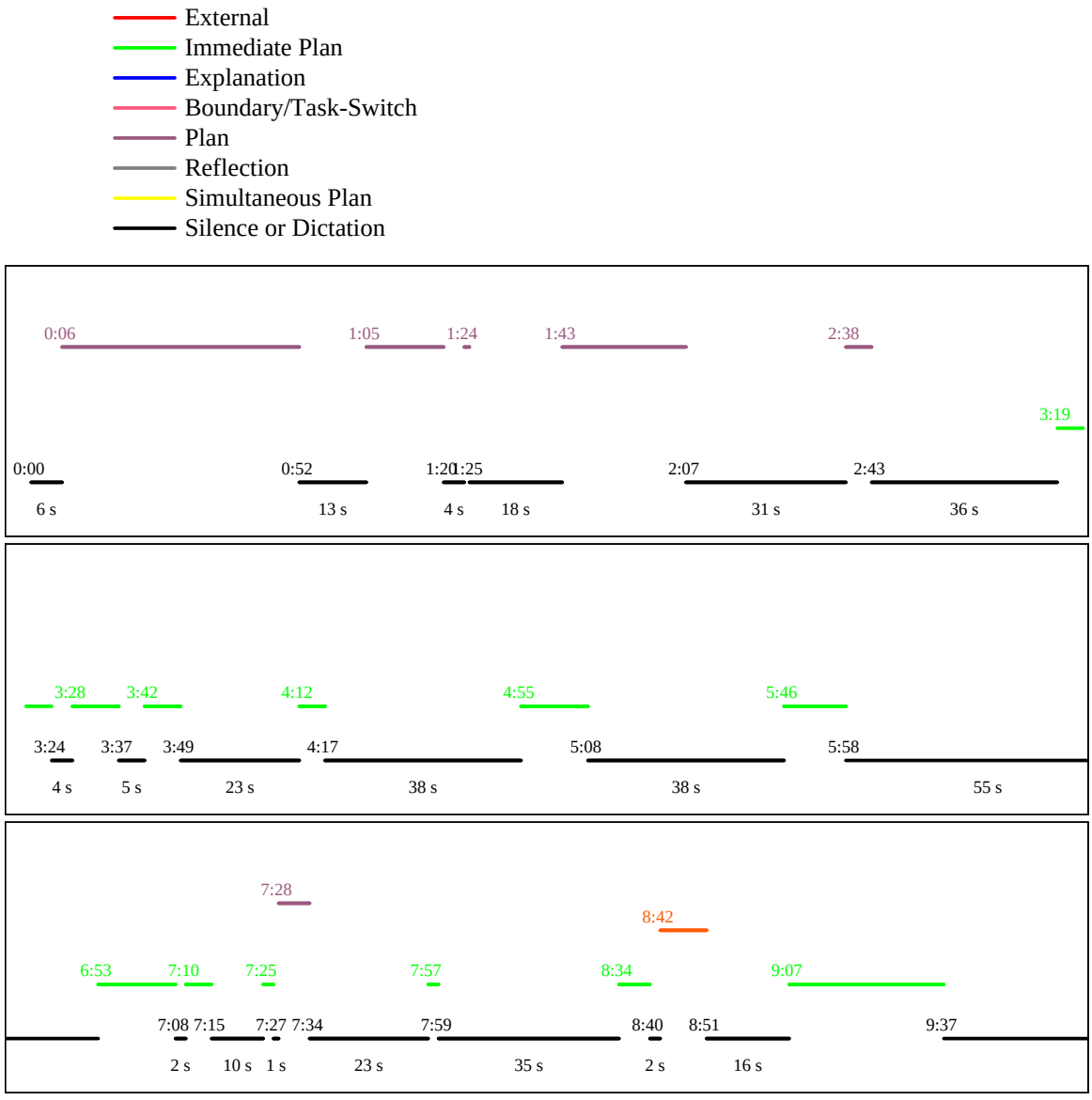

Figure 7: Subject 2's voice annotation timeline.

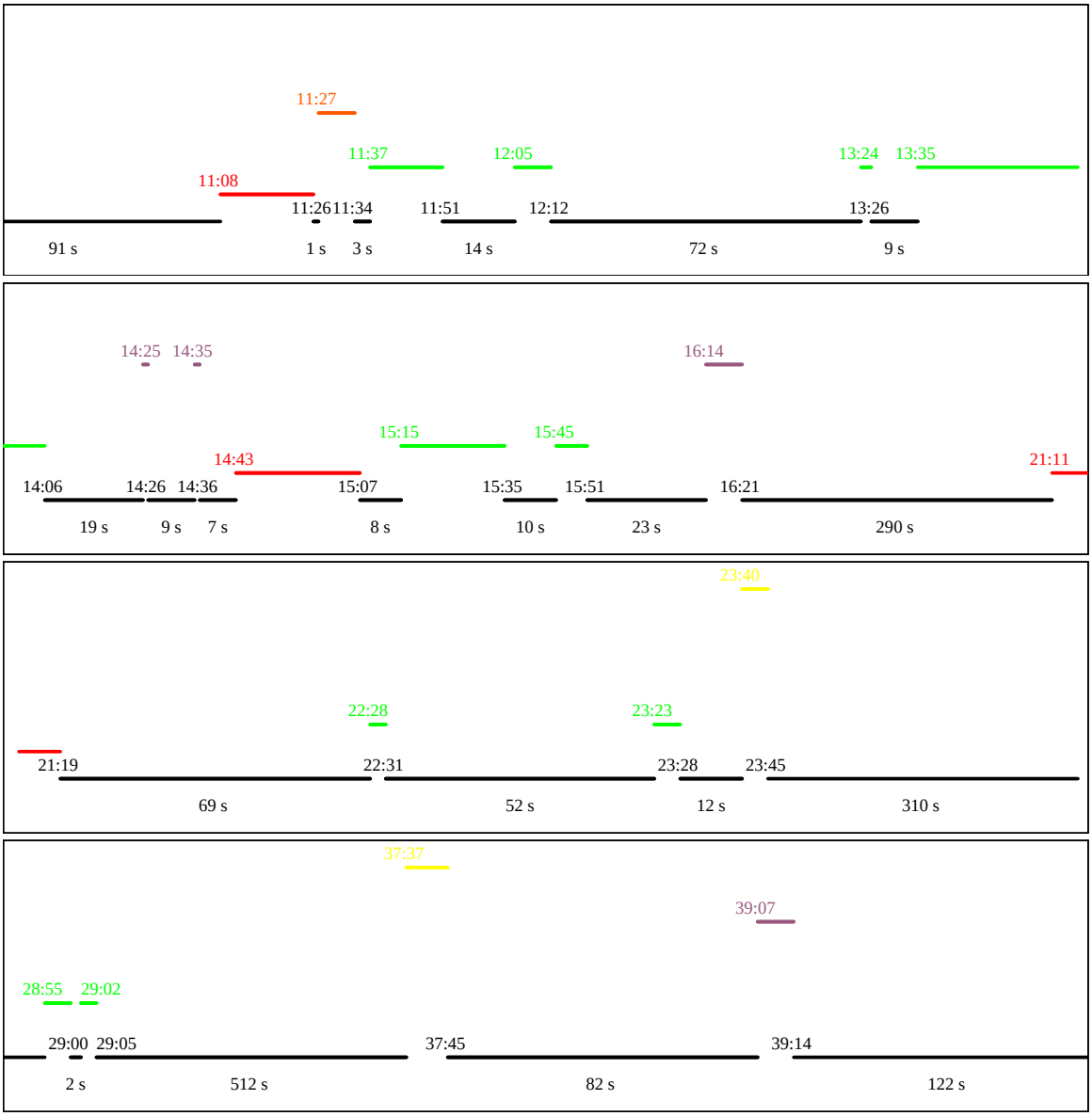

Figure 8: Subject 2's voice annotation timeline, continued.

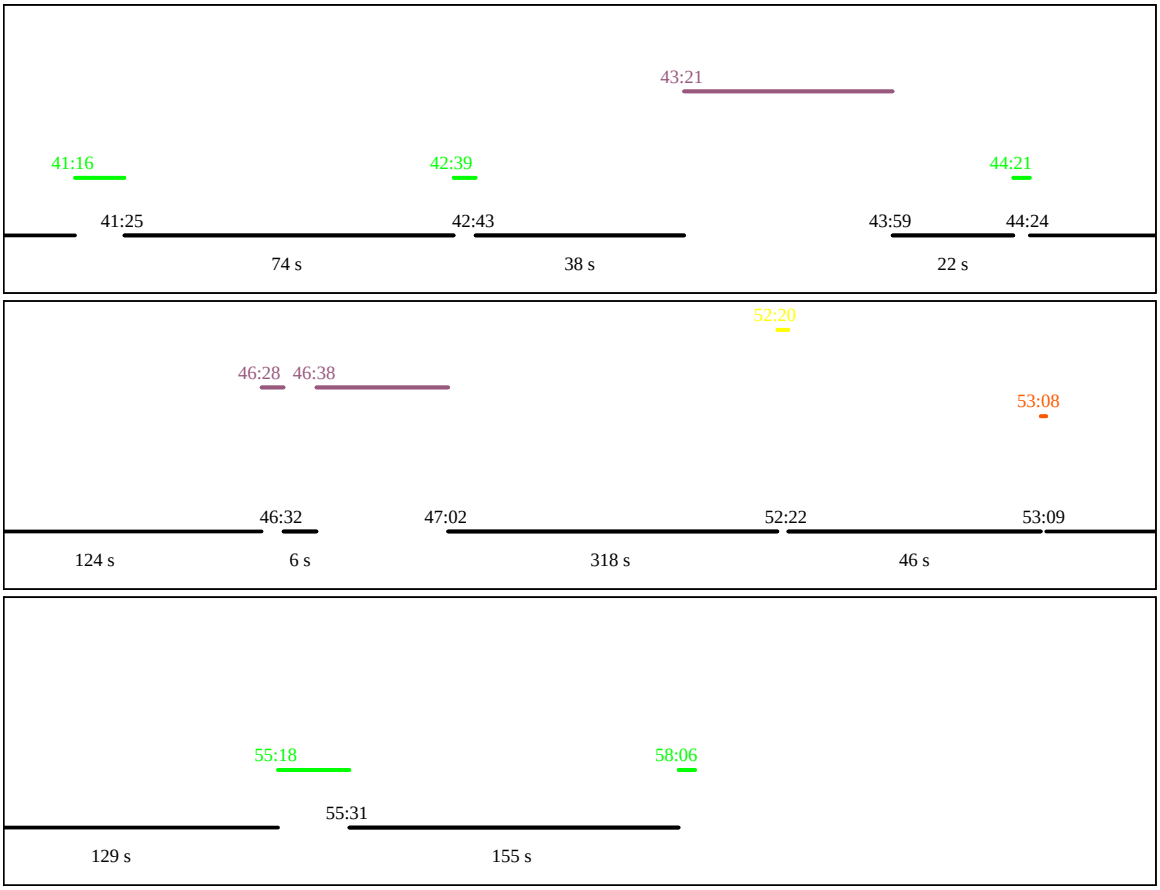

Figure 9: Subject 2's voice annotation timeline, continued.

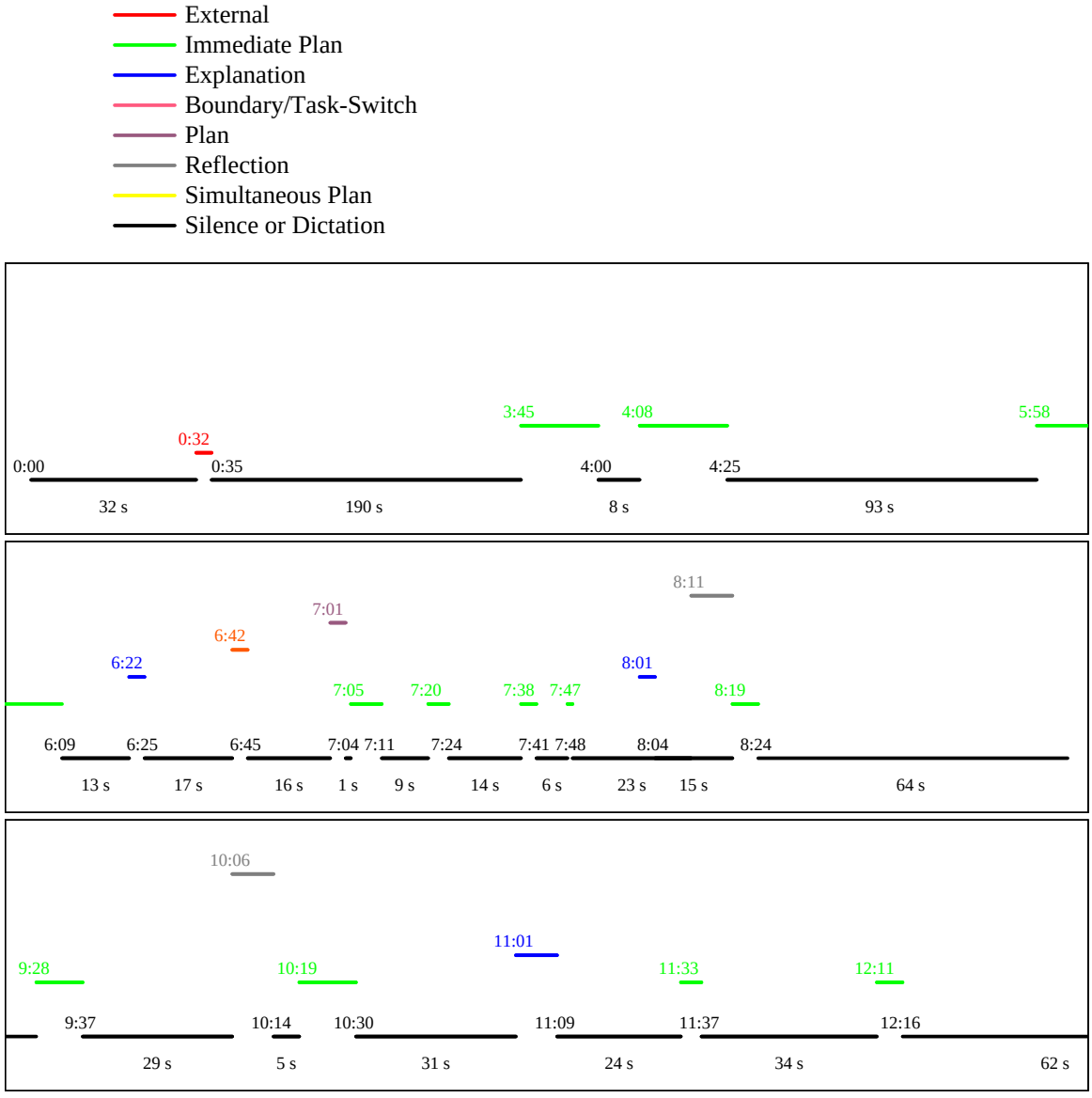

Figure 10: Subject 3's voice annotation timeline.

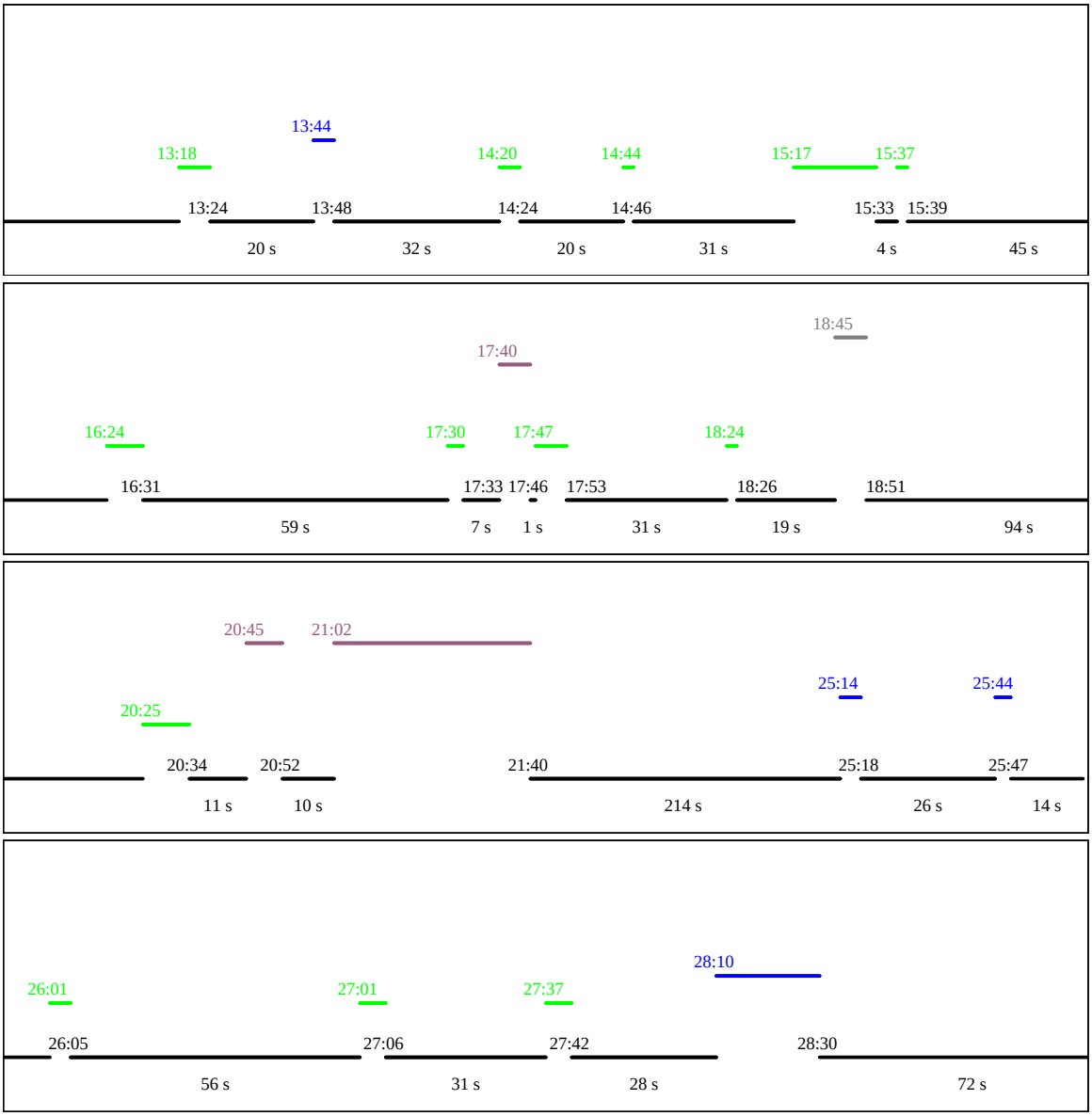

Figure 11: Subject 3's voice annotation timeline, continued.

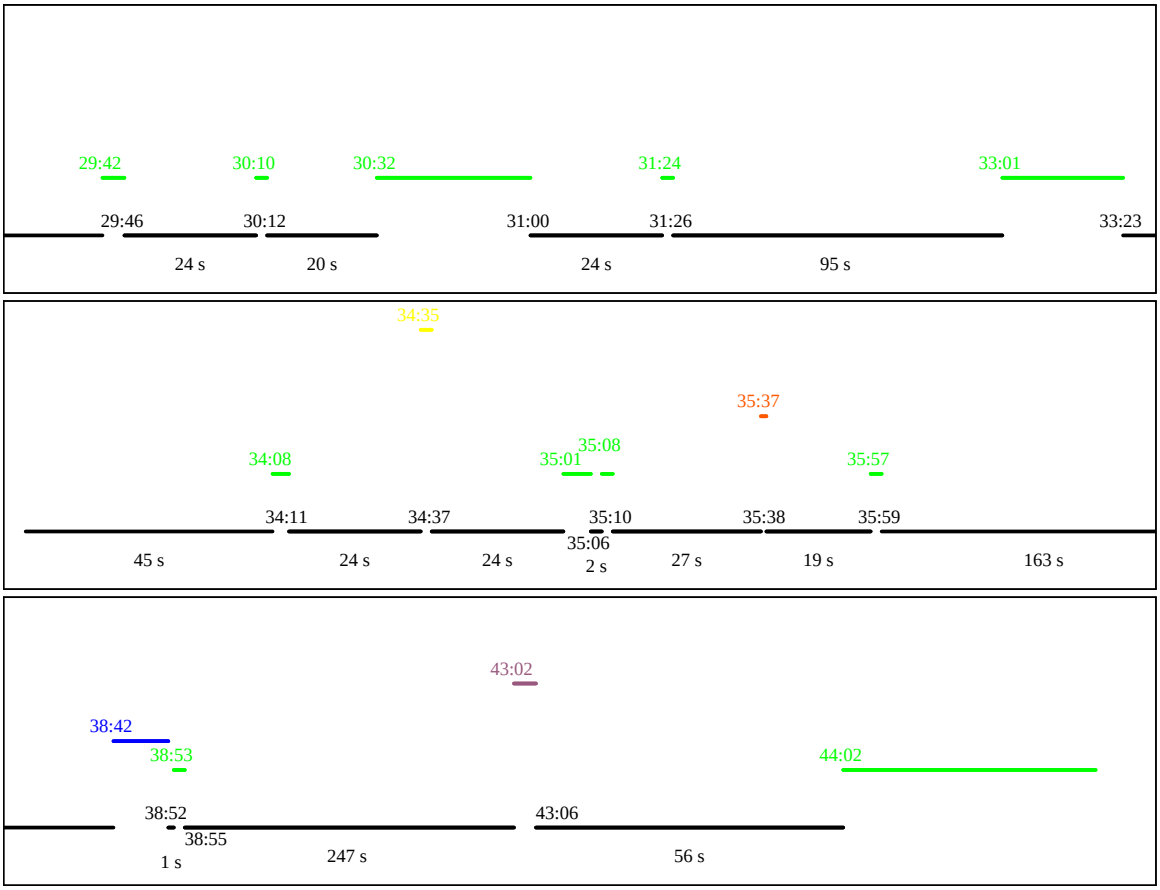

Figure 12: Subject 3's voice annotation timeline, continued.

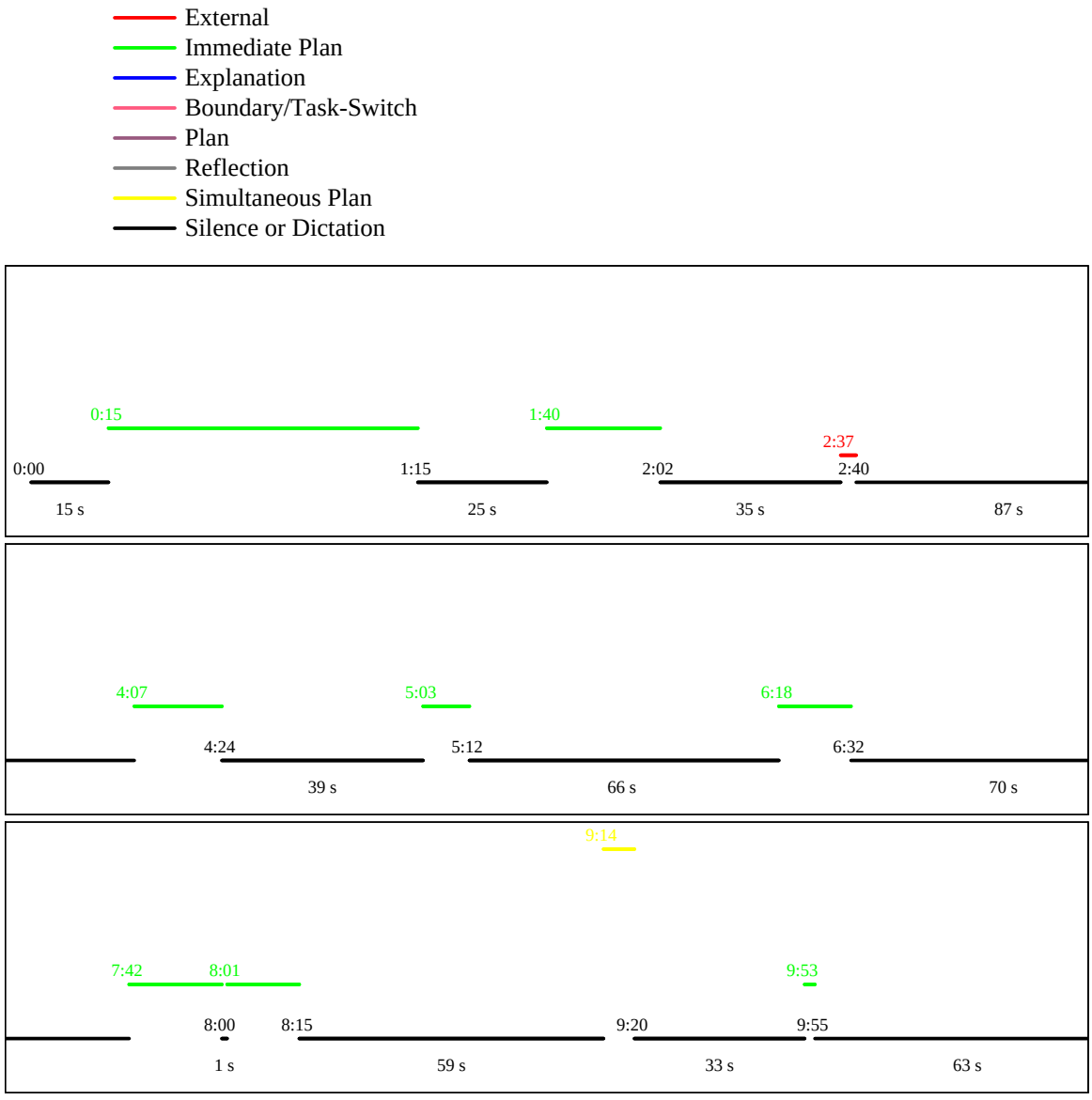

Figure 13: Subject 4's voice annotation timeline.

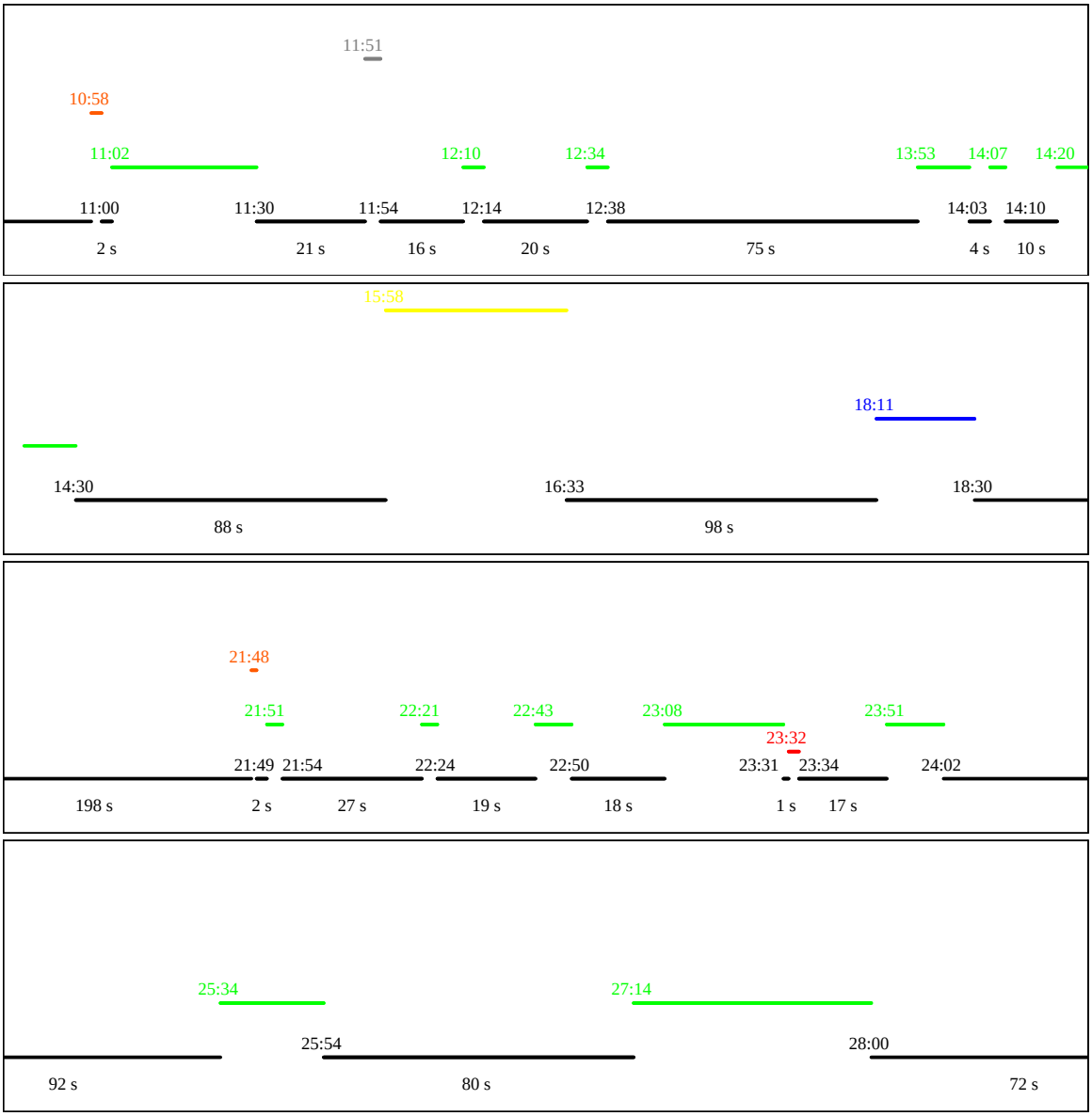

Figure 14: Subject 4's voice annotation timeline, continued.

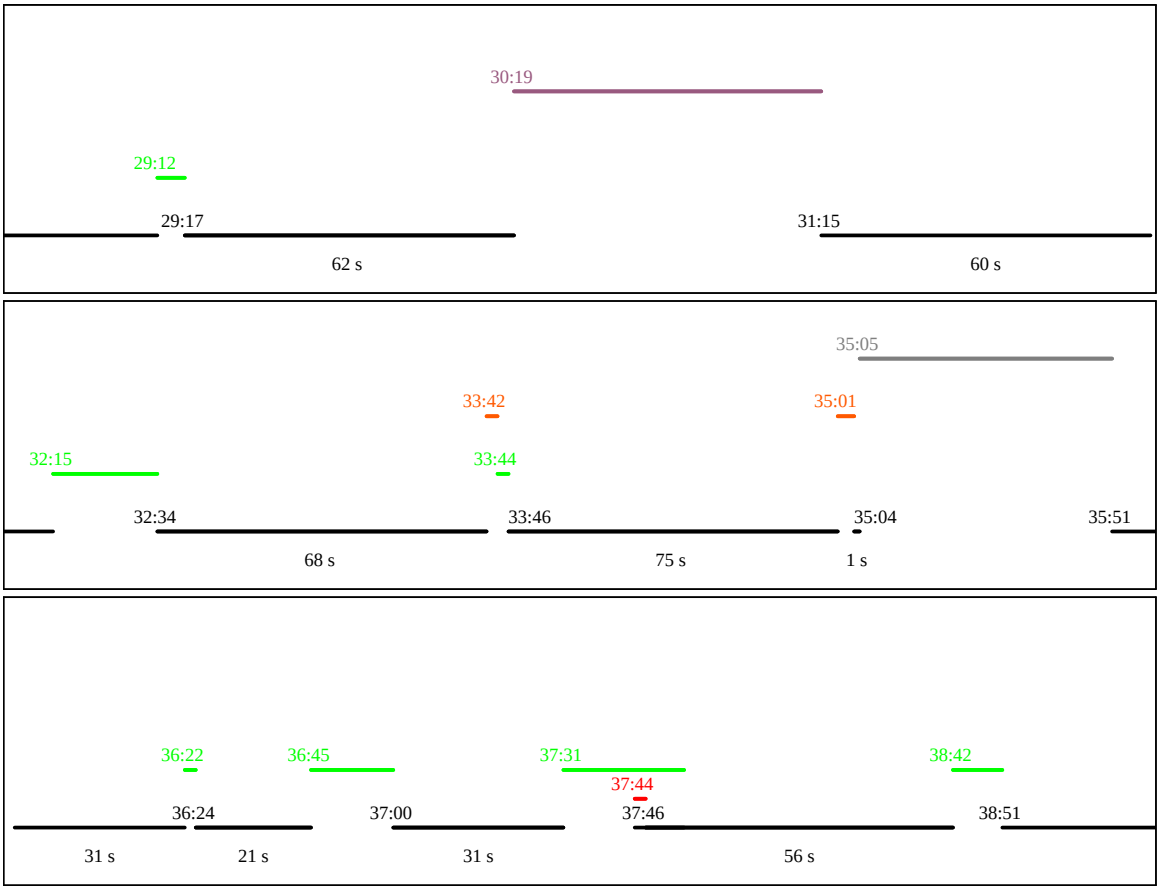

Figure 15: Subject 4's voice annotation timeline, continued.

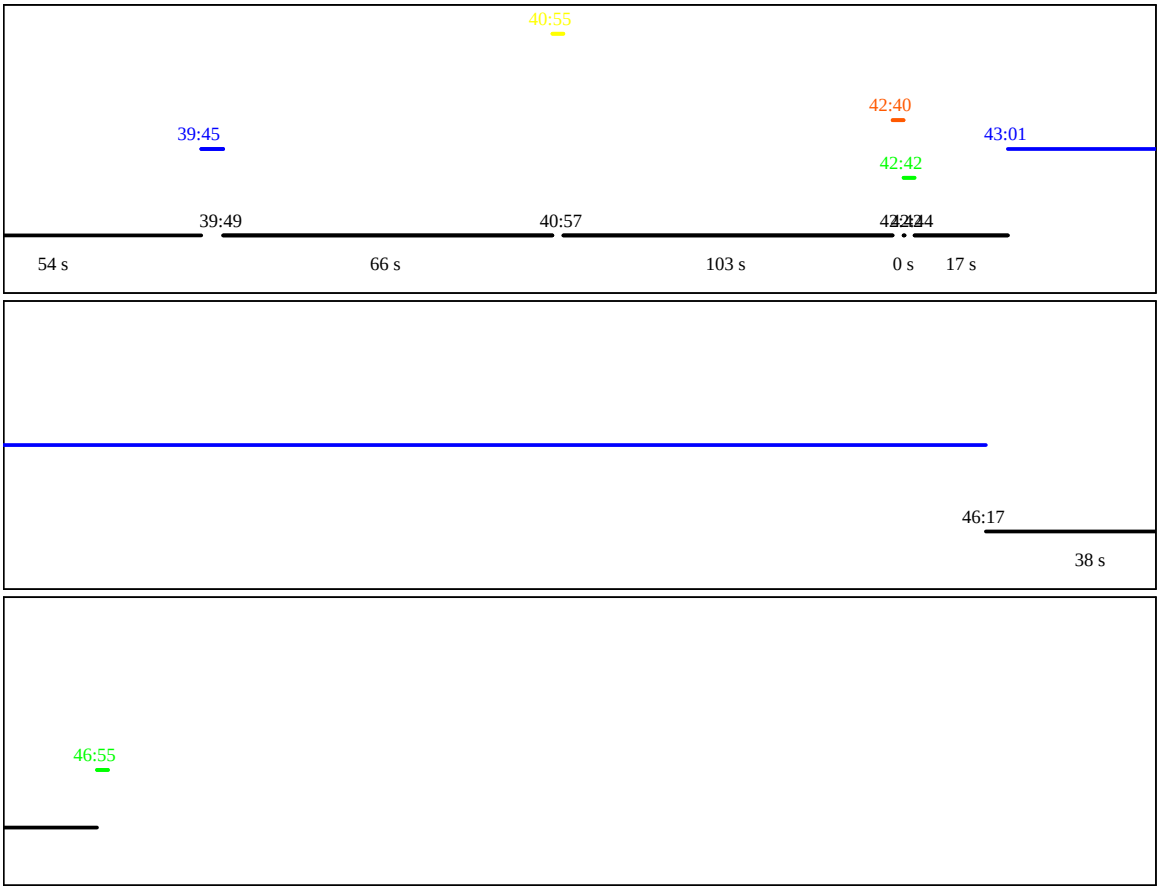

Figure 16: Subject 4's voice annotation timeline, continued.

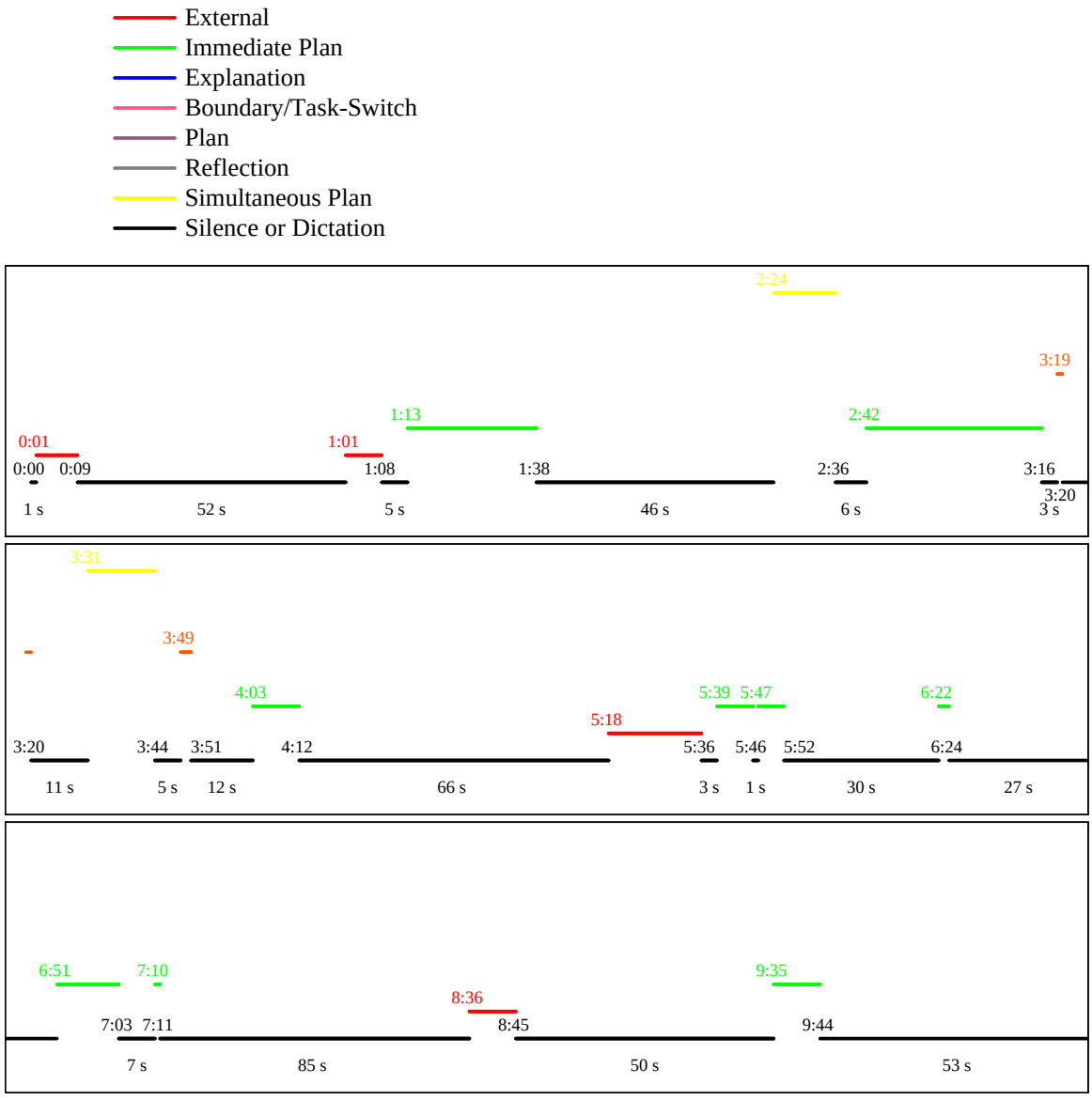

Figure 17: Subject 5's voice annotation timeline.

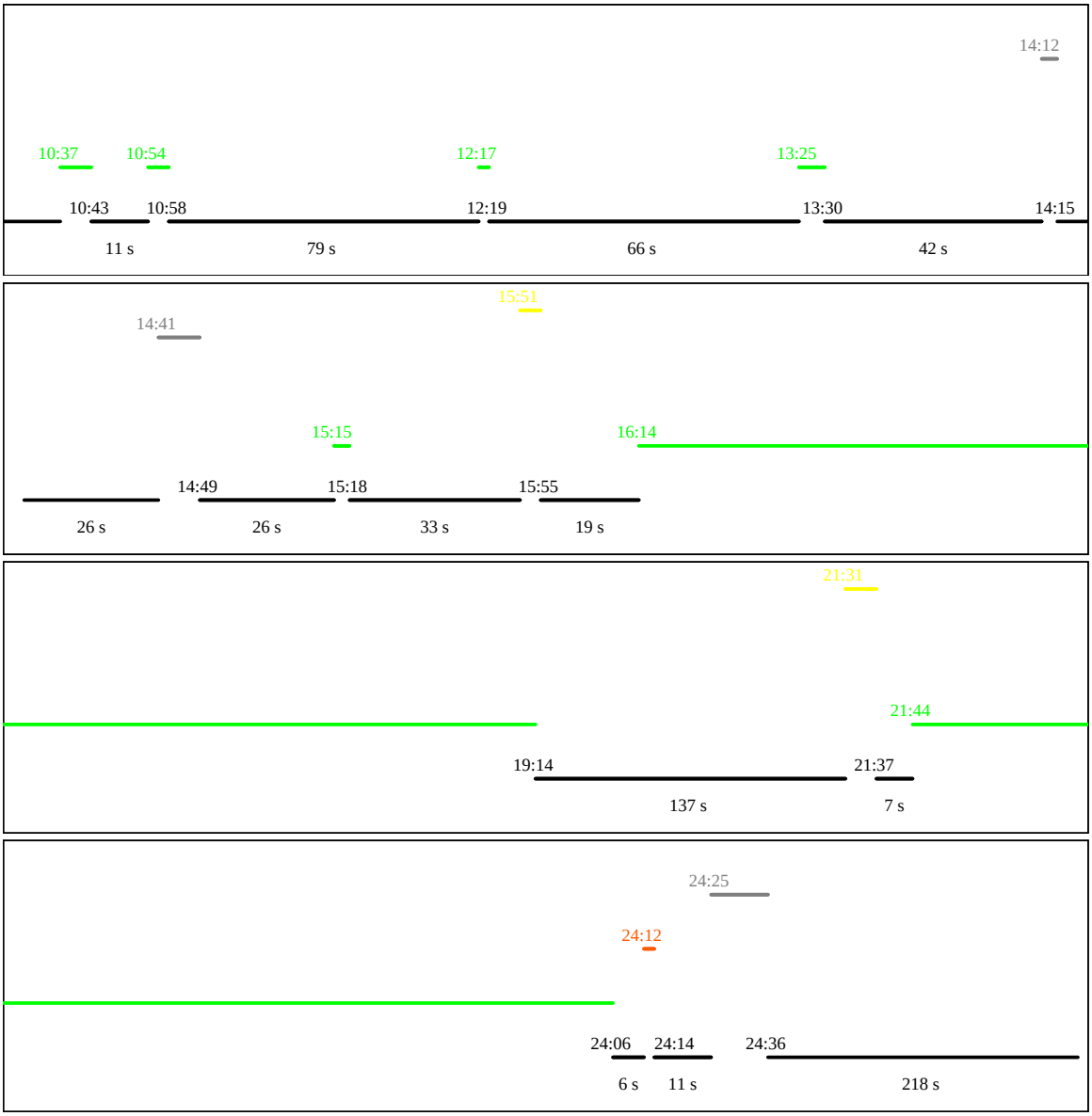

Figure 18: Subject 5's voice annotation timeline, continued.

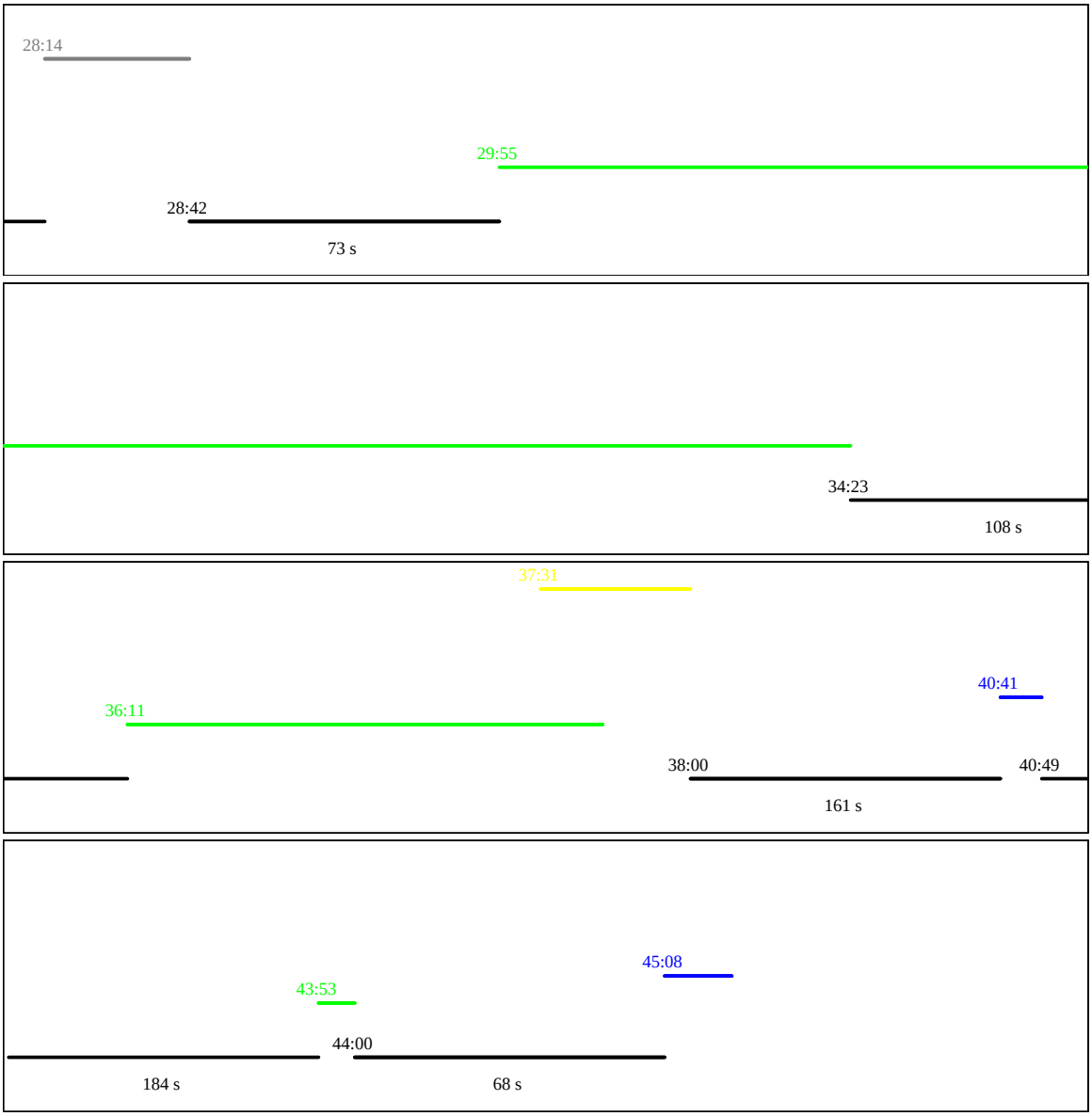

Figure 19: Subject 5's voice annotation timeline, continued.