

The Rhetoric in the Program

What Programmers (Don't) Say to Each Other In the Code

Clinton R. Lanier, Ph.D.

Assistant Professor of Professional Communication

New Mexico Institute of Mining and Technology

Abstract

This case study discusses the composition of particular elements of computer program internal documentation—imbedded source code comments. Through interviews, workplace observations and discourse analysis, five programmers were studied to identify different social and rhetorical factors affecting how these comments were composed. The study suggests that a common assumption—that comments are created specifically for collaborative purposes—is wrong. In fact, the programmers studied created comments more often for egocentric reasons. Possible factors contributing to the how comments were composed included the programmers' perception of their task priorities, their definitions of collaboration, and their identification of audiences.

1. Introduction

Behind every computer program there is an artifact known as the program's "source-code." This source-code is composed in different types of programming languages, such as C, C++, or Java, and kept in particular types of files. Whenever one opens any type of computer program, that source-code is called upon to deliver instructions for the computer to enact the specific actions of the program.

Outside of computer science or cognitive psychology little about computer program source code has been written. There are a few exceptions, however, one being Orr's 1999 study in which he identified computer programs as a particular type of genre. This is a significant claim, because it suggests that computer programs are less like a collection of mathematical calculations, and more like any other purposefully written document: subjective, open for interpretation, composed in a process of rhetorical decision-making, and shaped by the discourse community that created it.

Though not focusing specifically on how computer programs are composed, some scholars from within computer science agree with Orr by noting the vast subjectivity that programming languages allow. Sebesta, for example, points out the many different methods for creating even the simplest of functions (9-10). Specifically, he demonstrates that there are four different ways of adding two numbers together in the programming language, C. From these choices the programmer decides which to use and then composes that portion of the program. The source-code then, is a collection of these choices and rhetorical moves on the part of the programmer.

Within a computer program's source code is another rhetorical device, namely the source-code comment. Source-code comments (figure 1) are lines of text—similar to

author-written margin notes—imbedded in a computer program’s source-code.

Ultimately, computers only understand binary code, sequences of ones and zeros that equate to numbers, symbols and letters. Because binary is extremely difficult to write, read or understand, a number of programming languages have been created. Though these languages have vastly improved the understanding of source code and the ability to write complicated programs, they are still sometimes difficult to read and understand. Comments, therefore, allow programmers to insert messages in the source code, giving them outlets for communication. Whenever the source-code is “read” by the computer, these comments are skipped because they are preceded by symbols telling the computer to do so.

```
213  /* having verified that the proposed changes are legal,  
214     we now put them into effect. */  
215  if (pid < 0) {  
216     if (pid == -1) /* all procs other than current and init */  
217         ret = cap_set_all(&effective, &inheritable, &permitted);  
218     else /* all procs in process group */  
219         ret = cap_set_pg(-pid, &effective, &inheritable,  
220                        &permitted);  
221     } else {  
222         ret = security_capset_check(target, &effective, &inheritable,  
223                                   &permitted);  
224         if (!ret)  
225             security_capset_set(target, &effective, &inheritable,  
226                                &permitted);  
227     }  
228 }  
229
```

Figure 1. Example of comments and associated source code.

Figure 1 demonstrates typical comments found in source code. This particular section of code was taken from the Linux operating system’s source code written in the programming language, C. The comments are found on lines 213-214, 216, and 219, and are distinguished by the slash-asterisk combinations, which signify that they are comments and should not be ‘read’ when the program is enacted. The first comment,

lines 213-214, is often called a ‘header’ or offset comment and describes the following 15 lines. The comments in lines 216 and 219 are called inline comments and describe the code to the left of them.

These comments are important, because often people other than the original programmer read the source-code in which the comments are contained. Programmers use others’ source code for a variety of purposes. As Spinuzzi has pointed out, they will read source code to learn about unfamiliar solutions to problems or perhaps to understand the history of a program. Most often, other programmers read source-code because they are performing “maintenance” activities on the program. Such activities include updating the program, creating or adding new functions, or solving specific problems.

In each example—reading the code to learn, or reading the code to perform maintenance on the program—the reader is often faced with an unfamiliar document. Because, as pointed out above, there are different ways to create programs, commentary explaining how *this* particular piece of source-code works can be an important factor in helping a reader understand the program. Thus, better understanding of these comments could lead to better programming practices.

This article discusses a study intended to better understand how these comments are actually created in the workplace. By conducting a series of interviews with professional programmers complimented by brief observation periods, and by examining examples of their source code and the related comments, I was able to gain insight about the complex issues contributing to why comments are written, how they are written, and what they say. Drawing on literature from within professional communication and rhetorical studies, my findings suggest a network of factors—including an understanding

of task priorities, perceptions about audiences, and socially constructed programming practices —contribute to the creation and use of source code comments. Such an understanding of comments within the context in which they were created could lead to a better, more efficient understanding of programming practices for practitioners and scholars alike. To better situate the current study within existing literature, the next section discusses past and related studies on comments and on programming in general.

Commenting and Programming as Items of Study

Many studies have been conducted in the past either directly or indirectly relating to source-code comments and their place in computer programming. Having long been considered an instrument for helping programmers understand source code (see, for example, Brooks, A Theoretical Analysis; Brooks, Towards a theory; Schneiderman and Mayer), comments have been tested by numerous studies to find out how useful they really are. Some studies, like an experiment performed by Basili and Mills focus on the use of comments to help programmers understand never-before-seen code (which it did). Other experiments have focused on how the formatting of a program, or of the comments themselves, affects the reader's utilization of comments. Such a study (Deimel, Kunz, Makoid, and Perry) demonstrated that offset comments (the comment on lines 213 and 214 in figure 1), rather than comments on the same line as the associated code they were explaining, made a program much more readable. Further studies of typographical and formatting concern (Miara, Musselman, Navarro, and Schneiderman; Oman and Cook) have shown that attention to such formatting, including the formatting of comments, increases a reader's overall comprehension of the code.

Other past studies attended to the effectiveness of comments for simply increasing the understanding of the code. Experiments have given mixed results, indicating that very detailed comments do in fact help readers understand a program (Nurvitadhi, Leung and Cook, 2003), but at the same time make the program very difficult to read (Tenny, 1988). These findings are analogous to those from studies conducted on footnotes, which show that the added information in footnotes is in fact helpful to those who need the information, but also slows the reading process of documents (El-Sakran, 1990; Hartley, 1999; Jansen, Van Lijf and Toussaint, 2001).

Other studies have additionally demonstrated how comments aid programmers in understanding where items are (or should be) located within large programs—especially when those items are not located where the programmer intuitively expects them to be (Letovsky and Soloway; Soloway, Pinto, Letovsky, Littman and Lampert). So programmers, in this and other experiments, turn to comments in expectation of finding relevant and important information.

Studies not specifically focusing on commenting, but rather on issues of programming in the workplace have also contributed to our understanding of how comments are used. In particular, comments have been identified as providing programmers an outlet for representing ideas within the code. Such external representations are essential to the collaborative writing of programs because they allow various programmers/writers to more precisely inform one another of the very abstract ideas the program is designed to portray (Flor; Flor and Hutchins).

These representations, external to the code, are a form of communication between programmers (Bellamy). Hence these comments take on numerous roles in the

programming process. Spinuzzi found that comments contribute not only to the collaborative, shared understanding of complicated concepts within the code (as was found by Flor), but also to the training of new programmers, and to an organization's programming history. In none of these cases are comments used consistently or in the same way, but only sporadically and as needed in the process of programming. Spinuzzi even found one department (out of the three he studied) that resented the writing and maintaining of comments within the code, and so refused to do so. Such an attitude and lack of updated comments leads to program degradation over time and increases the amount of effort necessary to maintain a program according to a recent study on the corrective maintenance of programs (Kajko-Mattsson).

Hence, the perception of commenting is that it is an important programming practice, made evident by the fact that well over half of the software organizations surveyed in a study either had internal documentation (i.e., commenting) guidelines or had *partial* internal documentation guidelines (Kajko-Mattsson). The implication, then, is that industry recognizes the importance of source code comments and encourages their use.

Unfortunately, literature from within the computer industry also suggests that source code comments are still not used enough or are not used in ways that help unfamiliar programmers understand a program (Campbell; Raskin; Ray; Thomas; Zokaites). Programmers, it seems, do not often comment in such a way as to efficiently and effectively help readers of their code. One reason for this may be the absence of more socially and rhetorically-g geared studies in the collection of literature dedicated to

computer program documentation. A lack of understanding, in other words, about how to really write usable and effective comments.

The utilization of source-code comments, ultimately, is the use of language to communicate within workplace documents—use that is dependent upon social and rhetorical factors. Because these social-rhetorical factors—and not exclusively the technology—dictate the creation of a program (that is, the *writing* of a specialized document), studies directed at these factors could lead to better, more efficient programs and methods of programming. Professional communication is well situated to investigate how language is being used within programming because such an investigation can draw on many existing studies performed in the past on different, yet related topics. This study complements existing studies by using professional communication as a lens through which to study commenting as it is practiced in the workplace by professional programmers. The next section discusses the qualitative methodology used in a case study in order to better understand—from a social and rhetorical perspective—the creation and use of comments within one, small programming department.

2. Research Methods

Case study research was used in this investigation to understand the context surrounding the creation and utilization of source code comments within a real workplace. The study took place at a research laboratory located in the Southwestern United States. The laboratory—the Southwest Research Center (SRC, name changed)—employs over 400 people who work primarily as contract engineers and scientists for various United States Government agencies, as well for as private commercial entities.

Within SRC many professionals also work in a support capacity for the lab's infrastructure.

One particular department—the Security and Finances department—was responsible for maintaining SRC's complex billing and financial system. The previous system was based on the Model 204 database system, which had been originally purchased and implemented at this lab in the early 1980's. It was decided to move to a different system called BANNER, which has more online features than the previous system. My research was conducted during the move from the Model 204 database system to the BANNER system.

The case consisted of five participants (table 1). Each participant was in some way responsible for assisting in the BANNER upgrade. During the upgrade I interviewed each participant twice, observed the participants in the capacity of their respective jobs, and collected source code from each.

Table 1. Participants chosen for this study.

Participant	Background/Context
Able	<ul style="list-style-type: none"> • Professional programmer for ten years, at SRC for seven of those ten. Network Manager for all of SRC and direct supervisor of the other participants. • Directly in charge of the migration from the Model 204 system to the BANNER system
Joann	<ul style="list-style-type: none"> • Professional programmer for 20 years, all of it at SRC. Her current position is systems analyst. • Has been working on the Model 204 mainframe since its arrival at

Participant	Background/Context
	SRC and has built many of the programs that other programmers were trying to replicate in different languages for BANNER.
Simon	<ul style="list-style-type: none"> • Professional programmer for 15 years. • Fourteen years at SRC have mainly included programming mainframes in a variety of mainframe languages, including Model 204 user language, Rexx, SAS and COBOL.
Chuck	<ul style="list-style-type: none"> • Professional programmer for seven years—all of which have been spent at SRC. • Software developer, primarily responsible for working on individual projects as they are assigned (rather than working solely on the mainframe, for example).
Cooper	<ul style="list-style-type: none"> • Professional programmer for nine years total and for SRC for three of those years. • Security administrator, but in prior position created many of the new functions for the BANNER system that mirrored existing functions on the Model 204 system.

Between the months of April and June, 2005, I performed individual, semi-structured preliminary interviews with each participant. The specific purpose was to find out about the participant’s employment of source-code comments (especially how they may use comments and who they may write them for), to identify strategies that the participants use to help them write the source code comments (what is it that they may

have in mind when writing them), and to get general demographic information. For each, I scheduled the interview in advance and then met each individually. The interviews were tape recorded and later transcribed.

Observations of the participants were conducted at SRC during the months of June and July, 2005. I observed Simon, Chuck, and Joann individually for one hour each as they worked in the capacity of their everyday tasks, and generally during the course of the time I spent there. I observed Cooper, Able and Simon during a review meeting of one of Simon's programs. For each scheduled observation (that is, while they worked as I observed) I tape-recorded the session and took field notes. During my time throughout the study I took field notes of observations I made.

Each of the participants who created source code in the process of their work for SRC (4 of them—Chuck, Simon, Cooper, and Joann) were asked for source code representative of the type of code they would normally write. By looking at the comments they actually included in the source code, and sorting them into categories, I hoped to understand from a reader's perspective, the types, purposes and frequencies of the comments inserted by these programmers. The results of categorizing these comments would then be used to direct the questions in the discourse-based interviews (discussed later).

I collected a total of 9032 lines of code from Chuck, Simon and Cooper. Because the source-code given me by Joann was created over a period of 20 years and by a collection of programmers, I did not include it in this portion of the study (that is, the analysis portion), but did use it later for the discourse-based interview. Each of the programs collected were (according to the participants) programs representative of the

type created and maintained during the normal course of work at SRC. Each was created solely by the respective participant—meaning that the participant who gave me the program was the only programmer who actually wrote the code.

As the final portion of data gathering, in July and August of 2005 I conducted discourse-based/follow-up interviews with the participants at SRC. These interviews were conducted as focused interviews. After categorizing the source code comments from each (with the exception of Joann and Able), I presented to the participants the results of the analysis of the source-code and recorded their reactions to the findings, asking them specific questions about findings of the analysis. With regards to Joann and Able, I used as a catalyst for the question the source code given me either by Joann or by the others. Lastly, I solicited reflective accounts of why they made the choices they made in the source code.

I applied content analysis to analyze the source code supplied by Chuck, Cooper and Simon. Following guidelines outlined by MacNealy (1999) and Huckin (2004), I first selected a construct of study and then its associated artifacts, then I determined the units of analysis, and then I finally categorized the data to be analyzed. My purpose was to categorize, from a *reader's perspective*, the various types of comments included in the participant's source code. Further, the content analysis noted the location of the comments selected, and allowed me to compare what the participants said about comments (in the preliminary interviews) to the types of comments they actually composed. To identify general topics in order to understand the participants' views of commenting, the interview data was categorized according to the open, axial, and selective coding schemas as suggested by Flick (2003).

3. Results

This section provides the primary findings of my four data gathering methods: the preliminary interview, the observations, the content analysis of the source code, and finally the discourse-based interviews. The results are presented in the stages in which the data gathering occurred because each individual method leads to the next and so begins to create a unified understanding of the practices at this workplace. Following the results of each method I draw on findings from professional communication literature to discuss the complex interactions at work in this context.

Preliminary Interviews

During our first interview, the participants each had different opinions about the value of commenting (Simon and Chuck hated commenting and told me it was distracting, while the other three embraced it as a useful tool), but they each told me that source-code comments were primarily used for collaborative purposes. Specifically, as a whole they told me that comments were useful for communicating to someone else something about the program: an explanation of sorts.

Joann said that the reason to include comments in a program was “so no matter who goes in and looks at [the program] they’ll have a clue who did this and why they did it” (Joann, May). Similarly, Simon said that he includes comments “to explain what [I’m] doing...” (Simon, April), and Cooper told me that he includes them to explain “why I’m about to do what I’m doing” (Cooper, June). This was shared by Able, who said that he “will comment what the code actually is doing” (Able, June). Finally, Chuck suggested

that “at the top of every program there should be a comment of arbitrary length telling you what the heck the [portion of the code] is for” (Chuck, April).

It is interesting that each differed dramatically in how they chose to comment—some told me that they preferred comments at the beginning of portions of the code, also called header comments (Chuck’s preference), while others preferred comments on the same line as the code the comment discussed, also called inline comments (Simon)—but all suggested similar utilizations. Essentially they are suggesting that comments can explain how something in the code works, and why it was included. Collaboration, then—specifically to explain something—was the primary purpose cited throughout these interviews, and further implied by the participants when I asked for whom they wrote comments.

Joann said that she writes comments for “someone else,” because, she said “someone else needs to be able to read this.” (Joann, May). Again, agreeing with this sentiment were the other four participants. Cooper told me that when he’s writing a comment, he tries “to write it in such a way as if somebody else was reading it [so] that they would understand” (Cooper, June). Able simply said that his comments were “for whoever is going to be next to go in there and modify it” (Able, June), as did Chuck, who reported that he put “comments in there [the code] when it’s for someone else to read” (Chuck, April). Simon, lastly, told me that he didn’t often comment, but now that people were beginning to look at his code more often, he was “going to do it [comment] a lot more” (Simon, April).

Participants suggested, then, that they understood other programmers would follow their work at a later time and would need to understand how the program works.

To help new programmers understand the program, participants included source-code comments. Participants further suggested that the comments were written so as to be read by another person. Comments could serve as a method of training new programmers, and as means to explain how something in the code works, or why something in the code was there in the first place.

This is a powerful aspect of collaboration that inserted, written commentary can serve. Research from within computer supported collaborative work suggests that comments written by authors or readers of a document and inserted into the document, effectively turns the document into a collaborative tool. It makes the document function like a bridge through time and space between collaborators: it allows authors to “talk” to each other through the document (Miles, McCarthy, Harrison, and Monk).

This certainly extends to programming when considering the program source-code is the document in question. As previously pointed out, past research has shown source-code is used by programming teams to train members of the team in programming techniques, and in educating new programmers in the operation and history of the program (see, specifically, Spinuzzi). The participants in this study accepted source-code as a device for functioning in these ways, and suggested in the first interview that comments were a means to aid other programmers in understanding what they did and why.

As there are many different ways to arrive at the same solution in computer programs, even when using the exact same language, there is often reason to detail how a specific piece of code works (Kohanski). Further, because programmers will often implement different features of a language to do the same thing (Sebesta), and as

languages evolve to include different capabilities, there is a need to not only explain how something works, but also to justify its inclusion in the first place. Due to these reasons, and the fact that participants told me they utilized comments to overcome a language's subjective nature, I fully expected to find much attention paid to the composing of comments designed to help someone understand the code.

Observations

The only participant that inserted a comment during the observation stage was Simon. Interestingly, composing the comment was completed quickly and with little thought as to what it should say. This is in contrast to how he composed the rest of the program. For example, I observed and recorded him spend well over a minute trying to decide the name of a particular variable:

[Simon; speaking aloud while at the keyboard] Alright, do I have the right variable name? Source? Stock? Stocksource? Stock...mystock? mystocksource?... (Simon, June)

On the other hand, he wrote a comment and spent no time planning the comment or deciding what it should say:

[Simon; speaking aloud while at the keyboard] In the *if* statement I'll put...[types comment as he speaks] "delete from mysqltable to prepare for refresh value..." (Simon, June)

The comment was almost an afterthought. He did not go back over it, did not read it to ensure that it made sense in its context or at that location, instead he went on coding, with long pauses and much thought dedicated to the particulars of the code.

Likewise, slight attention was paid to comments during the meeting I observed between Able, Cooper and Simon. This meeting was a quick code review to look at the program that Simon was working on when I observed him earlier in June. Code reviews, or code inspections, are standard industry practices, in which the coauthors of a program, or at least those collaborating on the program and with enough knowledge to do so, read through a program to ensure its quality as judged by the reviewers and the organization's standards. Of the types of defects found in these inspections, the most commonly found—and perhaps the most important—are “minor defects” that do not actually affect the performance of the software. One of the most important of these minor defects, and one that increases the maintainability of programs, is the documentation, of which code reviews can drastically raise the quality (Porter, Siy, Toman, and Votta).

But little of the discussion during the meeting addressed any of the comments. Though Able did read the code, which included the comments, aloud, the majority of the time was spent talking about the file structure in use at this laboratory, the names the used for servers, and possible solutions to problems the program was designed to solve (Able, Cooper and Simon, July).

Neither Chuck not Joann wrote comments during my observations. Chuck was engaged in reading another programmer's source code in order to design a solution to a similar problem. Joann was reading the M204 Database code in order to find areas that needed to be upgraded in the transfer to BANNER. In Chuck's case the comments were turned to in order to find a basic understanding of how the program worked. However, he suggests that the comment was little use for him:

[Chuck; reading aloud from the sample program] “a vector of osg group pointers which is used to store the parents of the node’...What does that mean? Well I understand that but I still don’t see where the disconnect is occurring...(Chuck, July).

Joann had been working on the same code for over 20 years, and was intimately familiar with the location and meaning of nearly every function within that program. As she tried to locate certain items, she quickly read associated comments, and then moved on until she found the portions she was trying to locate. Hence, for her, the comments helped her find areas within the code, but did not add to her understanding (as she already understood the program so well).

Of note during the observations were two things. The first was Simon’s lack of any real planning that went into his comment when compared to the amount of planning that went into variable names, suggesting that the variable names were more important than comments. The second—and closely related to the first—was the similar amount of time spent on everything *but* the comments during the code review. In both episodes, I wondered why comments, though an obvious mechanism to help other understand a program, were overlooked or only briefly attended to while other elements were focused on. The next stage in my study was the analysis of the participant’s source code. This stage, I believed, would provide a more accurate view of the participant’s composing of comments than the observations would, as the observations only allowed me a glimpse of the participant’s activities while the source-code was a collection of their activities over a longer period of time.

Categorization of Source Code Comments

In all, I identified—from a reader’s perspective—10 different types of comments (table 2). This was unsurprising in that, as Spinuzzi pointed out, there could be *numerous* purposes for comments.

Table 2. Categories of comments identified in the participants’ source code.

Name of comment	Definition	Example
Functional Comments	<p>These comments identify the function of a line of code associated with it. They are placed in close proximity to the line of code and answer the question, “what is this,” or “what does this do?”</p> <p>For the reader, they identify the purpose of something by labeling it, or by explaining what the associated code does.</p>	<p>#ANIMATED TILE UPDATE #this handles row/col both</p>
Procedural Comments	<p>These answer the question, “how does this work?” They provide information about the operation of lines of code they are associated with. They will let the reader know relationships between functions, data used, or how an operation is performed</p>	<p>#receives single triplet list of len=3</p>
Historical Comments	<p>These provide the reader with some type of historical information about something in the source code. This historical data includes code</p>	<p>*/ String MySQLString = "SELECT survey_id, "</p>

Name of comment	Definition	Example
	<p>that is essentially ‘commented out,’ meaning that the programmer put marks around the portion of the code so that the compiler would ignore it.</p> <p>Historical comments can also include version and change date information, letting the reader know when something was changed and by whom.</p>	<pre>+</pre> <pre>* Created on</pre> <pre>December 13, 2003,</pre> <pre>7:28 AM</pre>
<p>Summary Comments</p>	<p>Condense an amount of code for the reader. For example, if there are ten lines of code that create a single outcome, summary comments can summarize the ten lines into one statement for the reader.</p>	<pre>PRINT HEADERS</pre>
<p>Endophoric Comments</p>	<p>Provide a cross-referencing between parts of the code. These comments will tell the readers where they are located within the larger file or software program they are looking at</p>	<pre>//end VerifyData()</pre> <pre>function</pre>
<p>Rationale Comments</p>	<p>These comments provide the reader an explanation as to why something is or is not in the code. These comments answer the question “why is this here (or not here)?”</p>	<pre>// This will make the</pre> <pre>database connection</pre> <pre>faster</pre>
<p>Highlighting</p>	<p>These comments provide visual highlighting for</p>	<pre>#+-----+-----+</pre>

Name of comment	Definition	Example
Comments	something within the code by standing out through syntactical marks.	+--+--+-
Further Research Comments	These note areas of the code that the author indicates revisiting for more work, or to fix something. They tell the reader that more work will be done to the area, or needs to be done to the area.	#This needs modified to handle mult/div
Evaluative Comments	Indicate areas of the code where the author has offered an opinion the programming style, or the operation of the code.	""deactivating: this works, but takes too long
Referential Comments	These comments refer to other parts of the program, including other files, classes or functions, and provide some type of association between the area of the code the reader is located, and the area that is being referenced.	#this function TuxTray->Submission ('getting' from Tux Tray)

But according to what the participants said during the preliminary interview, the two reasons for which comments should be included would be to tell someone else how something works, or to tell someone why something is there. I define these types of comments as *procedural* and *rationale* comments respectively.

In addition to the original ten categories of comments described in table 2, I noted 21 combinations of these categories (see table 3), resembling the findings of past studies regarding the types of footnotes used in academic articles (El-Sakron). In other words, while I found that comments could carry out one of ten functions, comments also combined these functions. For example, while a comment could describe the function of a piece of code (in such a case I would place this comment in the “Functional” category), it could also identify how the related code carried out the operation (in which case the comment could also be considered “Procedural”). In this example, I included the comment in a “combination” category: Functional-procedural.

Within these 31 total categories (10 individual and 21 combinations), I placed 739 source-code comments from 9032 lines of code (the comment could be multi-lined, and additional lines of code included code related, or unrelated, to the comment, which I did not categorize).

Table 3. Combination type and frequency of comments.

Type of Comment	Number Identified	Number Inserted by Author		
		Chuck	Simon	Cooper
Endophoric	19	2	16	1
Evaluative	1	1	0	0
Evaluative-highlighting	2	2	0	0
Functional	193	52	21	95
Functional-endophoric	5	1	4	0
Functional-evaluative	1	1	0	0

Type of Comment	Number Identified	Number Inserted by		
		Author		
Functional-evaluative-further research	1	1	0	0
Functional-highlighting	1	1	0	0
Functional-historical	2	2	0	0
Functional-historical-further research	1	2	0	0
Functional-procedural	3	3	0	0
Functional-procedural-summary	16	9	1	4
Functional-rationale	8	1	4	3
Functional-rationale-procedural	1	1	0	0
Functional-summary	229	58	5	94
Functional-summary-historical	4	3	0	0
Functional-summary-rationale	4	1	3	0
Functional-summary-referential	3	1	2	0
Further research	3	3	0	0
Further research-evaluative	1	1	0	0
Highlighting	41	39	0	2
Historical	168	134	15	5
Historical-functional	3	0	0	0
Historical-functional-further research	1	0	0	0

Type of Comment	Number Identified	Number Inserted by		
		Author		
Procedural	17	15		1
Procedural-further research	2	2	0	0
Procedural-summary	3	2	0	0
Rationale	3	0	3	0
Referential	2	2	0	0
Referential-functional-procedural	1	1	0	0

Of the 739 comments, what is most surprising is that almost half of them were attached, at least partially (meaning in some combination with another type of comment), to the Functional category. In other words, I found 477 comments that seemed to be, from a reader’s perspective, making a simple statement about the functionality of the associated source-code.

This is surprising because from a reader’s perspective, such comments say little about the code that would help when trying to understand the code. Instead of telling me how the code works or why the code is there, in essence *explaining* the code, all such comments really do is supply a somewhat obvious heading. Such would be the case of a simple portion of code stating “X + B” followed by the comment “Adds X to B.” This example does not tell me why the addition takes place or what either variable represents, but it only tells me the function of the code.

Further, as can be seen in table 4, all three of the authors who provided me source code composed functional comments (or comments with “Functional” as one component if combined with another category) more frequently than any of the other comments included. In fact, functional comments were included as part of a combination of comments more than any of the others (table 4). Only 54 total comments out of 739 had, as a component, procedure or rationale, the two purposes for which the participants told me they most often used comments.

Table 4. Frequency of each category in the combinations.

Comment Type	Frequency (including, and within combinations)
Endophoric	2
Evaluative	5
Functional	18
Further Research	6
Highlighting	3
Historical	6
Procedural	7
Rationale	4
Referential	3
Summary	6

These findings suggest that perhaps the participants themselves did not realize why or how they used comments, or were unsure how to include comments that were truly helpful from a reader's perspective. This became the focal-point of the discourse-based interview. The objective during this stage was to find out why they included so many comments other than the types they told me they included, and to find out why the comments were included in the first place if not for the reason they suggested during the first interview (that is, for primarily collaborative purposes).

Discourse-Based Interview: Simon

My discourse-based interview with Simon (similar to the observation) revealed how little he thought of comments when he composed a program. In the source-code he provided there were no header comments—the first comment was one created automatically when generating the code. The first comment that Simon had written turned out to be wrong, not matching the associated code (table 5). According to Simon, comment 2, “Creates a new instance of PopulateCalendar” was “a cut and paste” (Simon, August) that carried over from another program when he borrowed and then modified the code. The same is true for comment #3. Comment #4 is a commented out debugging line (that is, it was used to track problems in the program, but was flagged so as not to be run in the code). Comments 5 and 9—17 were the first he said he created (comment #6 also came from another program). Specifically, he told me that he created these to keep track of where he was in the code, as he was using a text-based editor with no function allowing him to find the beginning and ending of the program he was writing.

Comments 7 and 8 were commented out code—what I call historical comments—that he was not able to get to work at the time the program was due. Instead of deleting the code, however, Simon chose to leave the code in and revisit the functions to make them work at a later time (which he had not yet done).

Comments 18—27 were not his comments at all, but were already in the code that he borrowed from another programmer. Rather than delete the comments he chose to leave them in the code (notice the dramatically different style). Interestingly, comments 28—42 he said were not his style but he wrote them to stay consistent with the style prior to them (the style used by the author from whom he borrowed the preceding code).

Table 5. Comments written by Simon used for discourse-based interview.

Participant	Comment #	Comment
Simon	1	@param args the command line arguments
	2	Creates a new instance of PopulatePSLCalendar
	3	read data from the MySQL table and create an instance of the in the table
	4	System.out.println(sDate_time);
	5	end PopulatePSLCalendar class
	6	This PHP Page builds a data entry form for PSL Computer Usage.
	7	if (! session_id()) { session_start();

Participant	Comment #	Comment
		}
	8	if (\$Submit == "Submit this Form") {
	9	end error checking
	10	end Process after error checking
	11	end process \$Submit
	12	end if
	13	end session if
	14	close displayForm()
	15	close HTML_personnelnewemployee
	16	end VerifyData() function
	17	end isblank (s) function
	18	fields we want returned
	19	search filter
	20	connect to the LDAP server
	21	return an error message if we can't connect
	22	AD requires that the protocol version be set
	23	to keep from chasing referrals
	24	bind to the LDAP server
	25	return an error message if we can't bind
	26	perform the search
	27	store the results

Participant	Comment #	Comment
	28	count the results
	29	free the results
	30	close the LDAP connection
	31	if searched correctly, one account should return
	32	store the user's first name
	33	store the user's last name
	34	store the user's last name
	35	finalize the RDN and password for the user
	36	connect to the LDAP server
	37	return an error message if we can't connect
	38	AD requires that the protocol version be set
	39	to keep from chasing referrals
	40	attempt to bind as a means of authentication
	41	if the bind fails, the username and/or password are incorrect
	42	if the result count is not 1, there was a problem
	43	end function ldapAuth
	44	end VerifyData() function
	45	end for loop
	46	end isblank (s) function
	47	end function showLoginHTMLForm

Participant	Comment #	Comment
	48	Because of instance persistence, you should not declare any variables here... But the one below is okay.
	49	This will make the database connection faster
	50	end init method
	51	end doGet
	52	end doPost
	53	end destroy method
	54	end PSLTimecardController

Again, comments 41—45, and comments 48—52 were inserted for himself to understand where he was in the code while writing the program. Comments 46 and 47 were comments that I saw as procedural and rationale-based comments (comments expressly collaborative). These were the only comments inserted by him that matched his idea of what comments *should* be used for. When asked about these comments, he replied:

It's basically explaining to another java programmer why I'm doing what I'm doing. And so that is allowing them to know, you know when someone may come around and say 'well here's the problem, you put the connection out there and you should put the connection somewhere else.' And that actually is... to protect why I was doing something. (Simon, August)

With the exception of these two comments, most he inserted (eliminating the comment created by the compiler, the erroneously cut and pasted comments and the comments he inserted to make his code consistent with the code he borrowed) were for his personal reasons when he was actually creating the code. Those comments, the comments I call endophoric, helped him understand where he was in the code while writing it.

Discourse-Based Interview: Cooper

Simply put, the comments included by Cooper were almost all functional—that is, they really only tell the reader what the following function does but little else. Table 6 includes a selection of comments included by Cooper (he had many more comments than those included, but these are representative of the entire set). As can be seen all but a few are functional. The comments giving the most amount of information were the comment sets, 22—24. These comments, he told me, were specifically included for his colleagues, who had told him that they were interested in using the function—rather than the whole program the function was in—that the comments were associated with. To help his colleagues, Cooper made more elaborate and informative comments than he normally would.

Table 6. Comments written by Cooper used for discourse-based interview.

Participant	Comment #	Comment
Cooper	1	open connection
	2	program command object
	3	get records

Participant	Comment #	Comment
	4	close and release objects
	5	'cell row clicked 'cell col clicked 'cell row of cell being edited 'cell col of cell being edited
	6	hide right click menu items
	7	set toolbar customize and imagelist
	8	load toolbar
	9	set property because this button is pressed
	10	'Set btnFrs = tbarFRSReconcile.Buttons.Add ('ShowIgnored") 'btnFrs.ToolTipTexts = "Show Ignored Records" 'btnFrs.Description = "Show Ignored Records" 'btnFrs.Image = frmMain.imgStd.ListImages.Item("ShowPs") .Index 'btnFrs.Style = tbrCheck
	11	'load field array with display names -- index 0 'load field array with rs field named -- index 1 'load field array with field property values -- index 2
	12	program command object
	13	set normal and finalized tree view sql strings

Participant	Comment #	Comment
	14	get FRS groups
	15	set indentation for the tree view
	16	was the click outside the edit cell
	17	build insert statement from database record.
	18	'to add a row to the flex grid we need to separate the field data with tabs.
	19	get records from tbl_bofrs based on what node they selected and the view setting
	20	this function pads an amount to iPad digits with cWith and no point character
	21	'we know there is at least 1 decimal value because there was a decimal point. 'how many digits to the right are there?
	22	'sSecret is any string to be encrypted/decrypted. 'sKey is any string than is used to encrypt/decrypt sSecret. 'For ease of use declare a global variable like "gblsKey" with your key.
	23	'Get the ascii value for each char in key as we look through 'the secret string. 'Use mod and the key length to make sure we cycle through the

Participant	Comment #	Comment
		'key if the secret is longer.
	24	'Get the ascii value for each char in secret and XOR it with 'the ascii key value then convert it back to a char.

Cooper also included many historical comments (5 total), like comment 10, which like Simon consisted mainly of commented out code that did not function correctly and that he had planned on revisiting to fix (and which, like Simon, he had not yet done). By far, however, the majority of Cooper's comments resembled comments 1—4, 6—9, and 12—17.

Regarding these comments, I reminded Cooper of the answer he gave me in the first interview—that is, that when writing comments he tries to tell someone what he is about to do in the code. I pointed out that the comments he most often composed—functional comments—really only serve to identify the next line but give little information beyond that. At this he told me he actually wrote the comments before he wrote the code. For him, this is a way of planning or organizing the program before he begins coding: “as soon as I created the function and I knew what the calls were, I created the comments and then I'd go back and I'd make the comments work” (Cooper, August). Cooper told me that he does this because he does not want to spend the time creating the “flow diagram” that he was taught in school, and he also does not want to “just get in there and start coding” without any prior organization. To him, utilizing comments as planning notes is a form of “middle ground” between the two composing strategies

(extensive planning before the program, and planning during composition). (Cooper, August).

Discourse-Based Interview: Chuck

I lastly interviewed Chuck about his source code. Chuck's case was interesting in that during the preliminary interview he told me that, though useful for relaying information about the code to someone unfamiliar with that code, he does not like source-code comments. He regards comments as "clutter" and would rather spend more time creating "clear code" than creating lengthy or even inline, comments. What was surprising about this was that Chuck had included so many comments in the source-code he gave me: 341 in 4092 lines. Like Simon and Cooper, few of these, only 32 accounted for either of the uses the participants indicated during the preliminary interview (that is, procedural or rationale).

Chuck was the only participant to include any extensive header comments, such as comment 1 in table 7. I included comment 1 in a number of categories, including functional (it identifies the function), procedural (it describes how the class works), and rationale (it tells the reader why it behaves in such a manner). As a reader, this comment in particular would be very helpful when trying to understand how this particular class works and why it is needed. It also tells me its relationship to the program and what it provides. Lastly, it gives me a bit of history by telling me the latest update.

Interestingly, regarding this comment, Chuck said "I wrote it for myself. Certainly for myself" (Chuck, August). He told me that he wrote this particular comment while he was completing later portions of that code. He kept finding himself trying to understand this class' relationship to the rest of the program and how this class worked. He said that

after doing this a number of times, he finally decided to write a comment answering his questions to save him time later. So though another reader would find it useful, the primary purpose of this comment is not collaborative, but more egocentric: it is for the author.

Table 7. Comments written by Chuck used for discourse-based interview.

Participant	Comment #	Comment
Chuck	1	<p>Description:</p> <p>A Board is a group of generalized spots.</p> <p>This class should be used as a base class for various types of boards -- blackjack tables,cw puzzles,checkers -- any thing with fixed spots. The game pieces are attributes of the board.</p> <p>If you want a background image, blit that as background from the outermost game class, not this class, as this is a Group.</p> <p>Group base class provides:</p> <p>Add,copy,empty,has,sprites,remove,update</p> <p>Should not give both empty_spot_image and background_image; either/or.</p> <p>If background image, then we break-up the background image into default spot images. If neither, of course, then it's an <i>*invisible*</i> spot.</p> <p><i>*Derived*</i> classes set the location of the self.spots.</p>

Participant	Comment #	Comment
		<p>The way the images are passed to <code>__init__</code>, the look of the board (ie spot images) is still controlled from outside. No need to manually do anything unless get the spots through the <code>get_spots()</code> method.</p> <p>XC, YC are center of board -- boards all assumed array like</p> <p>Latest: submission is M*N array of invisible spots; <code>validate_submission(self.submission);</code></p>
	2	<code>#print 'check4guest:',m,n</code>
	3	<code>#print 'take_guestMN:',tile,m,n</code>
	4	<code>#this function TuxTray->Submission ('getting' from Tux Tray)</code>
	5	<code>#this function is boardspots</code>
	6	<code>#SPOT MAKERS:</code>
	7	<code>#change to background_tile</code>
	8	<p>#####This section was v2.5-v2.6a colored/weighted spots</p> <p>#####v2.7 the tiles (only) have various values.</p>
	9	<code>#NEW LOCALIZER</code>
	10	<code>#head @(m,n)</code>
	11	<code>#now remove "WC:"s</code>
	12	<code>#check above:</code>

Participant	Comment #	Comment
	13	#check below:
	14	#check left:
	15	#for ridx in range(head[0],tail[0]):
	16	#submission is a row
	17	#N=1,2,3 ~ singlets,doublets,tripplets
	18	#NOTE: still not chaining adjacent eqns...might leave off for now.
	19	#HACKHACKHACK!

This trend—commenting for himself—continued throughout his source-code. Comments 2 and 3, both debugging statements, were kept for the same reasons as his (as well as Simon and Cooper’s) code that had been commented out: it may be needed at a later time. Comments 4-17 were inserted for the same reason as comment 1: Chuck needed to understand at a later time what something meant without having to read the code:

I realized this is complicated...usually whenever you see a comment...most of the comments that are still going to be in here is stuff that while I was doing it, somewhere in the middle, I got confused somewhere in the middle, and went ‘if I’m confused now, I’ll definitely be confused later.’ So this requires a comment, that’s pretty much where those all came from. (Chuck, August).

Chuck also inserted many comments like comment 18 (which I identify as a further research comment). In this case, Chuck told me that the code was not doing something that he wanted it to do, yet he was able to finish the program without that particular functionality, however, finding that solution was still a goal so he inserted this note to remind himself of the problem.

Finally, Chuck also inserted comments that evaluated his own programming—comments that I call evaluative. Comment 19 is an example of such a comment. Again, this comment was included for himself, but in this case it was to remind him that his coding was, according to him, “less than perfect” (Chuck, August). This told him that if there was a problem originating in the area of this comment, he should look hard at how he created the associated function—it may be causing the problem. It also told him, in passing, to look for a better way to solve the problem.

For the discourse-based interview portion of the study I also interviewed Joann and Able. Because they did not provide me with source-code they wrote (Joann did provide source-code, but it was legacy code she co-authored/maintained over the years), I asked that they speculate on comments that someone else wrote.

Discourse-Based Interview: Joann

With regards to Joann, the comments in the code she supplied me are exemplified by those in table 8. As a reader, Joann told me that she found such comments—regardless of their length and placement—helpful. However, when I asked her what she thought was the main purpose of such comments—essentially why they were inserted—she told me that the reason was primarily egocentric. Specifically, she said that comments 2-5 were “mainly for the programmer” (Joann, August) to understand where he was in the code.

She told me that because of the complexity of the code in this particular language (Model 204 User Language) the programmers would often have to insert comments to let them know when subroutines ended (similar to Simon’s “End” statements).

Likewise, Joann told me that much of the commented out code, such as comment 7, was also probably kept by the original programmer to be reminded of things that were tried but did not work. Also, comment 10, which includes the initials of one of the programmers who created this particular code, was most likely inserted for the programmer who inserted the comment. According to Joann, comments like this were inserted “so that when something didn’t quite work he could go back in and see what he did to the code” (Joann, August). Again, the purpose is for the programmer and not for those who follow.

Table 8. Comments supplied by Joann used for discourse-based interview.

Participant	Comment #	Comment
Joann	1	This subroutine enables the clerk to quickly query their issues without leaving the program
	2	Go back to menu
	3	Print headers
	4	Go back to menu
	5	Print headers
	6	Default date is system date
	7	PF_KEYS = ‘PF KEYS ==> 1: IGNORE 3: END PROCEDURE’

Participant	Comment #	Comment
	8	Commence validation by subroutine
	9	Date verification
	10	&VLC
	11	LOC Verification
	12	Apply “ON” and “OFF” campus overhead rates

Again, the purpose is for the programmer and not for those who follow. Like Simon, Cooper, and Chuck, this was not the stated purpose during the first interview. Rather, she told me that comments were collaborative, useful for explaining something in the code to somebody else, not the original programmer.

Discourse-Based Interview: Able

The only person whose view of the purpose of comments did not change between interviews was Able. During my first interview with him he told me that he inserted comments for whoever was coming next—therefore they were a collaborative tool. Because he did not supply me with source code (he was primarily a group manager who reviewed the code of those he supervised), I showed him code from the other three who wrote code: Simon, Cooper and Chuck. For each he evaluated the code as a reader or maintenance programmer, rather than as a programmer himself. He had a very low opinion about Cooper’s type of commenting, and told me that it was too repetitive. He also had little good to say about Simon’s commenting, noting that the endophoric

comments that Simon inserted (i.e., “End if”) were of little use to him. However, both of the comments inserted by Simon and Cooper specifically for others (table 6, comment 48; and table 7, comment 22 respectively) were pointed out as being helpful for others. For Simon’s comment, he said that “because somebody modifying it [might ask] ‘Well why did he do that?’ Well this is why he did it” (Able, August). This type of comment he called “helpful.”

Also of note was that Able identified Chuck’s lengthy and very informative header comment (table 7, comment 1) as a comment inserted for others when Chuck himself admitted that it was only for his own purposes. Clearly, to Able, if the comment is “good” it matters little who it is written for.

Summary of Results

Primarily, I found participants believed commenting was a valuable, collaborative tool—the primary purpose for commenting according to much of the industry literature. Yet, none of them really used comments for this reason. Rather, comments were a tool principally used to help them accomplish their own goal of creating a working program, and none seemed to really understand that readers might have different needs than their own. The next section discusses the results of this study in light of professional communication literature in an attempt to understand why these participants utilized comments in the ways they did, and why they may have not seen the difference between their own use and their purported use.

4. Discussion: Task Priority, Audience, and Social Construction

Throughout the case, multiple explanations made themselves available to help interpret the results. Based on past studies and theories from rhetoric, composition and

professional communication, three particular directions seemed more appropriate than others. I noticed, first, a strong indication that the participant's views of their own task priorities contributed to how they chose to create comments. Also, their perception of what an audience is, and what secondary readers should already know, may also contribute to their utilization of commenting. Finally, participants suggested that their commenting practices were socially constructed through past experience, training or education. This section discusses each of these possibilities in turn.

Task Priority

This section discusses how the perception of the programmer's main priority when programming could affect how they commented. Couture and Rymer found in their 1993 study that a writer's "functional relationship to writing" and "the [perceived] importance of the writing task" are the "dominant factors which account for and explain how writers approach communication tasks in the workplace" (19). By surveying professional writers and professionals who write, they found that professionals who write will often only invest time in planning a document that they see is explicitly important. The perceived value of the writing task, then, affects the amount of planning they conduct for a document. Couture and Rymer specifically found that when writing a document they did not perceive as important, the professionals who write said that the aim of writing was the efficient completion of the task—and hence did not spend a great amount of time planning the document. The same could be true for Simon.

All but Simon told me during our second interview that the main priority of programming was to create the most effective product possible. Simon, however, said that the main priority of programming was "the efficient completion of a task" (Simon,

August). In other words, the goal of his programming was simply to solve the problem at hand as quickly as possible. Rather than plan out the program, he would rather “kick out some code quickly that actually looks good,” or at least get “something that looked like it would work” (Simon, August).

Further evidence of this belief was found in some of the program source code that Simon gave me as an example of his programming. When Able was reading the file it appeared to him that the name of the file, the comments included in the code, indeed the whole structure of the program, did not match what the code was actually doing. Able told me that his fear was that “this [program] is a skeleton, a template, and [Simon] just went in and replaced the contents and then left the [name of the program] up there” (Able, August).

As it turned out, Able’s fear was correct. Simon had indeed replaced the contents of an old file with new source code, leaving the file name and the comments and structure from the old program. He did this because “it was a one time program” (Simon, August) that he did not think he would ever have to revisit. The priority for Simon, then, was to just get the job done. This was in contrast to Able who was concerned with the future of the program, with the fact that someone else may have to perform maintenance on that program, and with the difficulty someone would have because of the mismatch between the information supplied by the file name, comments, structure, etc., and what the code actually did.

Though the other four participants told me that their priority when programming was to produce the most effective product possible, all intimated that at some level the most important thing was that the code worked. While maintaining the Visual Basic

programs he had been given when he began working at SRC, Cooper evaluated his predecessor's programming. He told me of one particular incident where the previous programmer used 26 lines of code for a particular function that could be completed with one line of code instead. Although he rewrote it with only one line to "make it more efficient," he admitted that "[the previous programmers] actually did a pretty good job of writing [Visual Basic] code to do what they needed it to do" (Cooper, August). In other words, it got the job done—just not as efficiently as it could have.

Further both Joann and Chuck told me that they understood the difficulty of going back and changing inefficient code because of the amount of time it may take: time, for both, is better spent creating new code and solving problems. Additionally, Able stated that he understood why more programmers did not have others look at their code before finalizing it, because "everybody else was busy, you know wherever you work it's like that" (Able, August). At some level, for all of them, the main priority is to make a program that works, not to help other programmers through better commenting.

Perception of Audience

This section examines the participants' views of audience: how they defined audience and how audience may influence their commenting practices. One view of the way these programmers look at audience would suggest that they lack the ability to "escape from a focus on one's own perspective," (Kroll, 179) and compose documents with others in mind. Further, and more importantly, this view would claim these authors are not avoiding "the 'egocentric' tendency to impute this perspective to others" (179). The importance of a writer separating herself from her own perspective—especially for texts that the writer knows may be used by someone else—is crucial, according to Kroll,

because “if a speaker or writer assumes that others already share his or her knowledge and views, then there is no need for clarity of expression, for elaboration of one’s ideas, or for detailed argumentation” (179).

However, the above view may be incorrectly applied to this situation, because all participants turned out to be very aware of other audiences reading their program, and suggested that when writing programs they would still account for those readers. I asked Simon if he ever thought about anyone else reading his program, and he said that “...I’m starting to do that now...I didn’t even think about that when I went to create the code. And when I would create the objects and things I didn’t think about that” (Simon, April).

Because the programs he currently works on are growing in size, more and more people are beginning to work on them—it is becoming “more collaborative” as he said. Thus, seeing this direct collaboration has influenced his perception of audience, and forced him to “come up with ways that are easier for somebody to pick up my code and just run with it” (Simon, April). Again, however, he is only coming to this conclusion from seeing evidence that people are looking at his code, it is the direct collaboration that he suggested earlier: “now that actually somebody’s picking up my Java code and looking at it. I’m starting to think about things like that [other readers]” (Simon, April). The implication is that when it is not obvious that someone will look at his code, he does not think about another audience aside from himself—he does not remove himself from his own perspective when writing.

Chuck, too, is aware of other audiences but only thinks about another audience when it is obvious to him that another audience will read the program. While he feels that the majority of programs he writes are only going to be worked on by him, he said that “I

do put comments when it's for someone else to read" (Chuck, April). He related an example to me where he wrote a program for Able. In that case, "before every section that I did something I put a big old bold uppercase header and said, 'Now we're going to do this, now we're going to do this,' so it's kind of a flow chart" (Chuck, April). Including such comments is not something he normally does. Other readers are not his priority, and he does not believe that other readers will look at the code anyway.

In contrast to the views of Chuck and Simon, Joann told me that she is always aware of other audiences, and when programming, she feels that "someone else is always going to have to go in here so someone else needs to be able to read this" (Joann, May). Thus, she continuously tries to keep in mind another audience. She credits this attitude to the fact that she has worked on and maintained the mainframe code for over 20 years, and has therefore had to read what others have written for that long. She said that there were many instances of programmers who did not share this attitude who felt, like Chuck, that "they were maintaining their own code, so why bother [commenting]" (Joann, August). Eventually, they would leave SRC and either she or someone else would be responsible for maintaining the code. In those cases, as a reader, she indicated that it was difficult for her to understand the programs they had written.

Interestingly, like Chuck and Simon, Joann did indicate she changes her writing style when explicitly aware of another audience. She told me that she was working with a colleague who was not that familiar with the Model 204 language; thus she "generated just a quick little ad hoc [program] for her to use as a basis for whatever she needed to look at." The amount of explanation in this program depended on Joann's perception of her coworker's knowledge of the language: "Because she doesn't really know the

language at all, I put a comment in here that says ‘in filename find without locking the records’ and then I put that in front of the ‘find.’ And then I put a comment that said, ‘now we insert the records,’ and then I put ‘fr stands for free directory’.” This type of compensation, she said, was “not something you’d typically do with people that knew the language.” Thus, as a programmer, Joann, like Chuck and Simon, measures the amount of knowledge she perceives her specific audience to have when she is explicitly aware of that audience (Joann, May).

Cooper, too, believes “other people are going to be working on my code, maintaining it” This belief contributes to the aid he tries to give those readers as a *secondary* audience. When working on his code, Cooper told me that “I just put in the comment, ‘I’m going to put this comment here.’ If I’m not doing that function right now I just put it there and move on,” thus thinking of himself as the primary audience while he is creating the structure of the program and utilizing comments as placeholders or “to-do” notes. But, he further says that “when I’m actually in the function, and I’m writing it...I think, ‘okay what am I actually trying to accomplish?’ And that’s my comment—I do think about how other people will read it.” So at the finer level he is aware of that secondary audience (Cooper, June).

Also as pointed out earlier, Cooper compensates for explicit audiences. When he is aware of a specific audience for something, he goes out of his way to help aid in their understanding. During our discourse-based interview, while he admitted that most comments we looked at together were for him, he talked about one particularly rich set of comments:

‘sSecret is any string to be encrypted/decrypted.

'sKey is any string that is used to encrypt/decrypt sSecret.

'For ease of use declare a global variable like "gblsKey" with your key.

This set, he said, was "for a future user," and that it was "really written for whoever's going to use this in the future." What makes this set of comments different from others, however, is that he explicitly knew that people were going to read these. He told me that, with regard to the function these comments are about, "other people were wanting to use my encryption function after I wrote it. So I made sure, and I knew that when I was writing it that other people were going to use it." He attempted to help this explicit, secondary audience, through these comments. He then admitted that if this audience were not explicit, he would not have created these comments: "So this is for them, and I normally don't do this. I normally talk to myself [in the source-code comments left in the program]" (Cooper, August).

Such awareness of explicit audiences as demonstrated by Chuck, Joann, and Cooper is not uncommon. Blakeslee demonstrated in her 1993 study of a group of physicists collaboratively writing a paper that authors will compose explicitly for specific readers. In her ethnography, the physicists solicited feedback for their article from readers and then used that information to make decisions aimed at these readers. The knowledge of readers aside from themselves affects the programmer's composing strategies as it did those of Blakeslee's physicists, and they compensate for what they perceive as the reader's level of understanding by inserting comments as aids. This is not unlike scholars who insert footnotes to bridge a perceived lack of understanding by their audience about topics in the text (El-Sakron).

Of the five participants, Able was the most concerned with an audience that includes more than just the author, primarily because, like Joann, his role often makes him that additional audience. When programming, he says that he tries “to be as cognizant about the other person as much as possible.” This is important, he says, because “eventually they will need to use the code or modify it.” His anticipation, then, is that others will read the code, and therefore he must help those readers understand the program. He maintains this attitude as a reader as well, and expects programmers to help him understand their programs. He says that programs should be “clear to read and understand,” so that “you can show somebody your code and they can understand it right away.” This in turn helps the reader “know where to make the modification.” In doing this, programmers are ultimately “making the program very efficient.” This efficiency is gained through an awareness of an audience other than the author—the people who will read and maintain the program in the future (Able, June).

While Chuck and Simon differed in their awareness of audience from Cooper, Joann and Able, all five had the similar assumptions about readers. In both interviews the participants suggested that readers need to have certain types of knowledge in order to read and understand the programs. In regards to how much information he includes for the reader, Simon stated that he does not like to include many comments because the code should speak for itself: “if the programmer knows PHP already, then...I don’t think you have to say any more” (Simon, August). The reader, then, is expected to be proficient in the language that the program is written in--although the level of proficiency is not mentioned, perhaps as much as Simon himself.

Chuck, likewise when discussing the amount of information he includes, told me that “if you don’t understand how I did that [program], well it’s not commenting that’ll help you.”. The assumption, then, is that readers will not only know the language, but know what an operation’s function is, why it was placed there, what its connections to other parts of the program are, and so on. Further, although he already established that his comments were primarily for him, he maintained that readers should be able to understand his programs with the information he does provide in combination with the code: “If I was to comment it more, there’s no need. If that was enough to write it, then it’s probably enough to understand it” (Chuck, April).

While Joann told me that “someone else is always going to have to go in [to the program] so someone else needs to be able to read this,” she also stated that “it’s typically someone who has fairly good knowledge of what they’re looking at; its not someone off the street who is not necessarily going to have a clue what that means” (Joann, May). Whoever is reading the code, then, will already know the language, perhaps already know the program and the organization.

This view was also shared by Cooper, who told me that although he does keep in mind another person reading his code, he does not “picture whoever’s modifying [the] code as a newbie” (Cooper, June). He considers them knowledgeable in the language that he is writing in—Visual Basic, and he acknowledges that the readers need to be at a certain level to understand his code. Further, Cooper, like Joann, suggests that whoever works on his code must have specific knowledge about certain features of the language. He relayed an example where he and Able were considering using someone else who already worked at SRC for help in creating and maintaining the Visual Basic databases:

We actually here in the building have found somebody--and they're very good at what they do--they've been programming [Visual Basic] for quite awhile. They're very good at programming it. But they were working in an engineering area where they did not deal with databases; they dealt with interfaces to circuit boards.

When we wanted to use them we decided not to because when we looked at their skill set and they saw some of our code, they didn't understand the database connectivity—getting records (Cooper, August).

People who work on the program, then, are expected to enter with the same amount of knowledge about the task (or the context of the task, including specific features of the language) as he has. This theme was reflected throughout the discourse-based interview, and in many instances it was suggested that the reader should have contextual knowledge as well as language-specific knowledge. I asked Cooper if a reader would know what “tbl_bofrs” meant in the following comment:

‘get records from tbl_bofrs based on what node they selected and the view setting

While referring me to a visual diagram of the table structure of the database, Cooper replied to me that “Every table starts with tbl_. So that is the business office frs records. This is table account, cost accounting, etc. So the reader would know what table blfrs is” (Cooper, August). The suggestion, then, is that readers must know the language, know the particular area of the language (databases), and know the structure of database tables at SRC in order to read and understand the program. Both Simon and Chuck had similar instances where they assumed the reader would already have knowledge of what specific things in the code mean. For Chuck, one example was the comment:

```
#get within some r_min, then sync centers
```

To which he told me that “anything that’s called ‘r anything’ specifies radius” (Chuck, August). This is his belief, and his understanding, which he assumes the reader will share.

Simon, likewise, used the comment:

```
// This PHP Page builds a data entry form for [SRC] Computer Usage.
```

Simon admitted that this was not enough information for someone who had just been hired to maintain the program, but he told me that he “wrote it for others working on the team,” and that it would give them enough information—his assumption for a particular audience (Simon, August).

Able—a team member—however, read the comment and suggested that it did not contain enough information for him to understand the code. “A data entry form for what?” Able asked, “What kind of form? What program?” Able further explained that “we have so many data entry forms it’d be hard to say ‘okay, which one are you talking about?’” (Able, August). Simon’s assumption that the reader would understand simply because the reader was a team member failed when tested by Able.

Collectively, these assumptions are the most compelling arguments for the view that the participants can not escape their own perspectives (Kroll). However, it may more easily demonstrate criticisms that Kroll includes in his discussion about audience. He states that while writers are told to think of a specific audience and then write for them (as opposed to writing for themselves), “in a good deal of writing the potential audience is so broad that the writer can make only very general assumptions about the readers” (175). Therefore it may be the case that when writing a program without the knowledge of an explicit audience (for which the participants do demonstrate specific rhetorical

strategies), the programmers make the general assumption that readers of their source-code will have a certain amount of knowledge to make their own task of writing easier. Thus, a novel audience for which to write would be modeled after themselves. In a case like Chuck's header comment, appreciated by Able, such a model would be correct.

Social Construction

Lastly, it was most interesting to learn how the participant's past experience and training influenced their methods of commenting. The social construction of their styles in and beliefs about commenting reflect back to the respective education they received and the discourse communities of which they were a part. Education especially was influential to Cooper, Able and Simon.

Referring back to table 6 of the participants for this study Cooper had the most unique and consistent commenting style. Before each line, or every few lines of code, he inserts a comment that identifies what the next few lines of code do. Upon reflection—in the discourse-based interview—he said that he wrote these comments before he wrote the code as a way of outlining the program. But in the preliminary interview—in which he had first told me that he includes the comments for others who may maintain his code—he told me that his commenting style was like the style of his graduate advisor: [My graduate advisor] had a big influence on my programming style. And he put his comments before his code... And maybe, subconsciously, I picked that up." Thus, situated learning—what Freedman and Adam call facilitated performance—had a strong influence on how Cooper commented. Further, Cooper told me that in his programming classes, the instructors often told him that commenting was an important part of programming: "They [the instructors] talked about it [commenting]. They said stuff

like...’you need to comment [because] other people are going to edit your code’” (Cooper, June). This attitude reflects Cooper’s own attitude about why he should comment.

Able, too, implied his education, and his training in comments during his education, was reflected later in his professional programming experiences. He told me that within his programming classes commenting was treated as an afterthought and not as important elements in the code. He said that instructors discussed commenting “in a way such that they understand...people document the code after it’s done.” This affected how and why the students included comments: “so in a sense you’re documenting so that your grade improves” rather than for maintenance (Able, June). Likewise little thought was paid to commenting in his later jobs, where, he said, they “never actually sat down and reviewed the code.” All that mattered in these positions was that “it [the code] works and the program looks good.” Interestingly, though Able is now a manager and concerned with maintainable programs with quality comments, as pointed out in the section regarding task priority, the above attitude is still somewhat implied in his first interview.

In my discussion with Simon, past training and work experience were both suggested as influential to how and why he comments. Regarding his formal education, he said that within his classes, “they were so militant on the comments that if you—it doesn’t matter if you created the best software on earth...if you didn’t have the comments that they wanted, then you would get marked down on those comments” (Simon, June). The commenting requirements were so extensive as a matter of fact—“three or four pages of comments,” detailing “what the system did, what the system was supposed to

do”—that including them became “more of a drag because its becoming more of a class requirement than something you do to help others.” (Simon, June).

Further, Simon told me that such strict focus on commenting practices was a “waste of time,” since “every place you go to has their own ways of doing things.” For example, at his first job out of college he worked for a very large petroleum company. In that setting he said that, “they already had the structure of the comments they wanted, so you already had the bulleted list of when you’re supposed to have comments in there.” However, at SRC, he said that “we’re all on our own” with regards to how to comment, meaning that there were no set conventions in place and accounting for the varied styles exemplified in the participants’ source code. But even without a unified, organizational convention, SRC has still contributed to Simon’s *programming* style.

During part of our first discussion Simon told me about how his arrival at SRC affected how he programmed. Specifically, he was telling me that before he began working there he used to (due to programming conventions at the organization he previously worked for) comprehensively plan the programs he created. At SRC, however, he stopped doing so because, he said, “if they [the programmers already working for SRC] were working on an assignment they would just start coding...and that’s kind of what I did, I’d just start coding and come up with something.” Thus, Simon quickly mimicked the styles of programmers already working for SRC.

Finally, within his code, I found evidence of Simon changing his commenting style to match the way others commented. Referring again to table 5—the examples of Simon’s source code comments—he said that comments 18-27 were written by another programmer and included in source code that he cut and pasted for his own use. Rather

than delete the comments he wrote example 28-42 to match. This is only a singular example, but notice how different these comments (comments 18-42) are from the others that Simon himself wrote. Commenting styles, then, can be quite varied depending on who has written them, and those same styles can be shaped by the context in which they are written.

Chuck said simply that he had not received any training, and that he developed his commenting style over time (SRC was the first and only organization he had worked for). He said now that he tries to comment as little as possible, but that six years ago he was “commenting a lot.” The difference, he suggested between his style now and six years ago was simply his level of expertise as a programmer. He told me that six years previous he was “experimenting with things”—implying that he was still learning—but since then he doesn’t have to comment anymore because he has learned to make his code so clear that comments are unnecessary. (Chuck, April).

Joann had the least formal training, consisting only of a few programming classes, and much time spent at SRC learning by helping to build the original Model 204 Database. She said that she could not recall any commenting training in school or at SRC. Hence, whatever commenting “style” she has, she says she gained through her work at SRC, and specifically through collaborating with other programmers. When she began there in the early 80’s, she said they “didn’t have any rules, as in this is what comments will look like and stuff,” and so she and her colleagues “kind of started mimicking each other’s [commenting] process.”

She indicated that this mimicking—actually the process of creating a house style—was not at all formal, instead of discussing how to comment, she said that “I think

we just did em [that is, developed commenting conventions as they were needed].” She relayed one particular account where a certain programmer included no formatting within the source code, and hence all of the lines of code were flush left with no spaces between lines and no significant visual differences between the code and the comments.

So in that case we finally said, ‘you know I don’t think so, let’s talk about making some little silly asterisks [and] set some rules down, so that those of us a little older didn’t die from going blind.’

This type of collaboration transformed into the commenting conventions used at SRC throughout the development and creation of the Model 204 Database. These conventions were written down and included in a 3-ring binder—the official, organization-specific Model 204 User Database Guide—distributed to all existing and new Model 204 programmers at SRC. The text about commenting is brief, and reads:

Using Comments

Use comments! However, do keep them concise. Comments must be preceded by at least one asterisk (*).

It then goes on to give a simple example of the type of comments one should use. Thus, meetings and conversations about commenting, as a result of collaboratively working on the program, evolved into an official (if not simple) convention about commenting for the Model 204 Database at SRC.

From within professional communication, it is not surprising to see how the commenting styles of these participants may have been shaped by past experiences and discourse communities—that is, socially constructed—because so many past studies on writing in the workplace have demonstrated the amount of influence these factors have

had on shaping writers. What is surprising is how none of these participants share a view of commenting. Because all of them currently work in the same discourse community, and often collaborate with each other on documents, it would be easy to assume that their commenting styles and beliefs would at least somewhat resemble one another. Such was the case found in Spinuzzi's study of three different programming sites. Members of the three sites held shared views about how and when to use comments, and about the value of those comments. Perhaps in the current case, programmers did not interact enough or work collaboratively enough to develop such a unified view.

5. Conclusions

This case study suggests very interesting and important aspects about the ways source code comments are actually composed and used in the workplace, and further explores beliefs and perceptions held by the participants that may contribute to their utilization of comments. First and foremost, as suggested in other studies, the participants used commenting for a wealth of purposes. Often, comments were used for a more egocentric reasons, namely to help them complete their assigned duty of creating a working program.

This use is certainly not invalid considering the complexity of computer program design. As documents, computer programs are lengthy, intricate and dense textual artifacts. Hence, utilizing comments as planning and outlining mechanisms, or as reminders and providers of contextually relevant information, should be completely natural uses for a device as convenient as comments. Such uses have been noted before by researchers such as Flor or Bellamy. But what I also found was that the participants *could* write comments very useful for collaboration—for later maintenance—when they

set out to do so. They know what kind of information other programmers need when code is being read for the first time, and so are able to assist new readers through their comments. However, as those in industry are already aware, this happens all too infrequently. In this small study, only two comments were really written for other programmers, meaning that the comments were well thought out to provide the necessary information for unfamiliar collaborators in the best way possible.

Participants most certainly draw on a network of complex factors in order to compose computer programs (of which commenting is a component). By using professional communication as a lens through which to view our discussion, three possible factors became evident. The first factor was the priority that the participants placed on their various programming tasks. As with professionals who write as a component of their job (but whose main task is not writing), well thought out and planned composition, thereby making more maintainable code—of which commenting would be a major element—is secondary to their primary task (which in this case is simply to make a working program).

Thus, creating effective commentary useful for collaboration is an afterthought once the main priority—that is, the working program—has been accomplished. An effective strategy, then, would be to write comments useful for creating the program, but that is also useful for maintaining the program. Such a strategy was attempted by Cooper, but according to Able his commenting style was less than helpful for someone unfamiliar with the program.

Additionally important are the perceptions these participants held with regards to audience and how much knowledge that audience must already have before reading the

source-code. Thinking about one's audience is always difficult. It is especially difficult when thinking of an audience for a document not often thought of as a document in the traditional sense (after all, it's a computer program, who will even read it?). For programmers I'm sure it is even more complicated to understand and possibly compensate for a reader's potential knowledge to help them understand the program, considering all of the different knowledge domains a reader must have in the first place: contextual knowledge, language-specific knowledge, knowledge about the problem and traditional solutions, and knowledge about the way "it's" done there, to name but a few.

Hence, when the participants want to model the audience for which to write comments, they are faced with a complicated decision concerning which knowledge domain(s) to consider. As suggested previously, perhaps the right choice is simply to model themselves as the reader, which allows them to then write comments aimed at someone like them. When thought is put into such comments, this strategy becomes an effective one, exemplified by Able's appreciation of the header comment that Chuck wrote for himself.

And finally, the influence of the different discourse communities of which they were/are a part is displayed throughout their programming (and commenting) methodologies, and probably likewise affects the first two factors, task priority and the perception of audience. For some of them, like Simon, Able and Cooper, their training through education influenced their later commenting views. Simon now dislikes commenting, and spends little time thinking about or planning the comments he creates. In our discussions he suggests he picked up this distaste in commenting from his instructors and what he saw as their overzealous attempts to make the students comment.

Able, on the other hand, suggested that in school commenting was viewed as secondary to the code, and he further states that this point of view was carried over to later, professional employment. His view is different now only because his position as supervisor forces him to think of the long-term maintenance of the code. However, as pointed out in the section about task priority, his current view is still influenced by his past education. Cooper, perhaps, was most influence by his education, even coming to the realization in our interview that his commenting style may be taken directly from the style of his graduate advisor, with whom he closely worked.

Both Chuck and Joann suggest that they learned on the job, and discuss how their current workplace and past projects have influenced their programming/commenting styles and beliefs. Also worth pointing out is that each had different training and educational experiences, and that each now exhibits different views about commenting as well as different commenting styles. Had this organization written and enforced a standard, uniform commenting style such differences might not have been found.

Throughout is suggested that programmers may be unaware of their own use of comments in practice. During our first conversation the participants seemed to genuinely believe that they were writing comments a certain way and for a certain purpose. However my observations showed that they gave little thought to comments, especially when compared to other facets of programming. And likewise my discourse analysis of their comments, from a reader's perspective, showed that they really wrote and inserted few comments of the kind they suggested they should include. It was not until I pointed out how they were commenting did they suggest they used them for other purposes. Though I only interviewed five programmers, this may not be uncommon—how often do

we really examine our own programming or composition practices? It may be worth the time to better understand how we are using comments in practice to get a better sense of how we can improve those practices.

Conducting a workplace-based case study was important because it gave a better sense of the social and rhetorical factors connected with how these programmers compose. What we are left with is a better idea of not only what some of those factors are—such as questions about task priority, awareness of audience and secondary readers, and the social construction of commenting and programming styles and perceptions—but also how they may interact to influence what the programmers write. The study suggests that a great deal of thought should go into the composition of internal documentation to make it useful to the long term maintenance of a computer program, however (as in this case) this may not occur often enough.

This study holds implications for practitioners and researchers alike. Practitioners of professional communication may add value to their positions within computer programming organizations by helping their programming colleagues look at ways to better write programs. Because professional communication has traditionally investigated many of the issues suggested by this study (such as collaboration and audience awareness), professional communicators are well positioned to study how programmers could increase their effectiveness in these areas. Professional communicators are often trained in identifying strategies to better understand potential audiences and including the informational needs of future readers. Examining a program's internal documentation to assure that it accounts for future readers seems like a natural task for professional communicators.

Professional communication practitioners can also assist programmers in structuring comments for those future readers. As suggested by the findings in this study, programmers may believe that their comments actually serve a different purpose. While they may believe their comments are telling a reader why something is included in the source code, they may only be identifying the function of a piece of code. Because professional communicators often review text and documents from a reader's perspective, they can assist programmers in writing comments designed to help readers better understand the program. By extension a new literacy may be developed and implemented: a literacy of internal documentation.

Researchers can use these findings as a starting point from which to plan and conduct future studies. As the findings here (as well as the findings by researchers like Spinuzzi, Flor and others) suggest, qualitative studies provide a wealth of detail about the strategies that programmers use. More studies could be planned that take place in the context in which programs are created. Valuable studies could include a qualitative investigation into the extent to which programming resembles collaborative writing. Researchers could focus on such issues as collaborative planning, review, feedback, and composition.

Researchers can also use the implications of this study to investigate the composition strategies of programmers. As my findings indicate, there may be various methods that programmers employ when composing programs. More research could be conducted to discover methods that work better than others, and to understand what makes certain methods better than others.

Finally, this social and rhetorical-based cross-disciplinary study provides a different way of thinking about programming. As a subject, computer programming is often misunderstood. To many it is an intimidating and mysterious practice based on complex mathematical formulas. In truth, however, it is a complicated social practice within a network of community-defined rhetorical activities. In fact, the act of creating computer programs, from the smallest element like a comment, to the entire program, is remarkably similar to the creation of any document. Studies such as this one could do much to aid in our understanding of programming. And since so much of what professional communicators do takes place in the computer technology field, such knowledge would also do much to help these professional communicators in practice, and help scholars and teachers in the field and classroom.

References

Able. Personal interview. 28 June 2005.

Able. Personal interview. 10 August 2005.

Able, Cooper and Simon. Observation. 12 July 2005.

Adkins, Mark, Reinig, Jeannette Q., Kruse, John, and Daniel Mittleman. "GSS Collaboration in Document Development: Using Group Writer to Improve the Process." Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, Maui HI: IEEE, 1999, 11.

Basili, Victor R., and Harlan D. Mills. "Understanding and Documenting Programs." IEEE Transactions on Software Engineering 8.3 (1982): 270-283.

Baecker, Ronald M., Nastos, Dimitrios, Posner, Ilona R. and Kelly L. Mawby. "The User-centered Iterative Design of Collaborative Writing Software." Proceedings of the 1993 International Conference on Computer Human Interaction, Amsterdam: ACM Press, 1993, 399-405.

Bellamy, Rachel K. E. "What Does Pseudo-Code Do? A Psychological Analysis of the Use of Pseudo-Code By Experienced Programmers." Human Computer Interaction, 9 (1994): 225-246.

Brooks, Ruven (1981). "A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs." Proceedings of the 1982 Conference on Human Factors in Computing Systems, New York: ACM Press, 1982, 125-129.

Brooks, Ruven. "Towards a Theory of the Comprehension of Computer Programs." International Journal of Man-Machine Studies, 18 (1982): 543-554.

Burnett, Rebecca E., and David Clark. "Shaping Technologies: The Complexity of Electronic Collaborative Interaction." Computers and Technical Communication: Pedagogical and Programmatic Perspectives. Ed. Stuart A. Selber. Greenwich, CT: Ablex, 1997. 171-200.

Campbell, Neil. "Thirteen Steps to Better Documentation." Computing Canada, October 1996: 28.

Chuck. Observation. 5 June 2005.

Chuck. Personal interview. 14 April 2005.

Chuck. Personal interview. 10 August 2005.

Cooper. Personal interview. 16 June 2005.

Cooper. Personal interview. 10 August 2005.

Couture, Barbara, and Rymer, Jone. "Situational Exigence: Composing Processes on the Job by Writer's Role and Task Value." Writing in the Workplace: New Research Perspectives. Ed. Rachel Spilka. Carbondale, Ill: SIU Press, 1993. 4-22.

Cross, G. A. (2000). Collective form: An exploration of large-group writing. The Journal of Business Communication. 37:77-100.

Cross, Geoffrey A. "The Interrelation of Genre, Context, and Process in the Collaborative Writing of Two Corporate Documents." Writing in the Workplace: New Research Perspectives. Ed. Rachel Spilka. Carbondale, Ill: SIU Press, 1993. 141-152.

Deimel, L.E., Kunz, L., Makoid, L., and J. Perry. "The Effects of Comment Placement and Reading Times on Program Reading Comprehension." Proceedings of the 19th Annual Conference on Information Sciences and Systems, Baltimore: The Johns Hopkins University Department of Electrical Engineering and Computer Science, 1985, 595-601.

Eastwood, Alison. "Firm Fires Shots at Legacy Systems". Computing Canada, January 1993: 17.

El-Sakran, Tharwat Mohamed El-Sayed. "Footnotes in Academic Written Discourse: A Formal and Functional Analysis." Diss. University of Wales, 1990.

Erlikh, Len. "Leveraging Legacy System Dollars for E-business". IEEE IT Pro, May/June 2000: 17-23.

Fjeldstad, R. and W. Hamlen, "Application Program Maintenance: Report to Our Respondents." Tutorial on Software Maintenance. Eds. H. G. Parikh, and N. Zvegintzov. Washington DC: IEEE Computer Society Press, 1983. 13-27

Flick, Uwe. An Introduction to Qualitative Research. Thousand Oaks: Sage, 2003.

Flor, Nick V. (1998). "Side-by-side Collaboration: A Case Study." International Journal of Human-Computer Studies. 49: 201-222.

Flor, Nick V. and Edwin L. Hutchins. "Analyzing Distributed Cognition In Software Teams: A Case Study of Team Programming During Perfective Software Maintenance." Empirical Studies of Programmers: Fourth Workshop. Ed. Jurgen Koenemann-Belliveau, Thomas G. Moher, and Scott P. Robertson. Norwood, NJ: Ablex, 1991. 36-59.

Freedman, Aviva and Christine Adam (1996). "Learning to Write Professionally: 'Situated Learning' and the Transition from University to Professional Discourse." Journal of Business and Technical Communication, 10 (4): 395-427.

Gebhardt, Richard (1980). "Teamwork and Feedback: Broadening the Base of Collaborative Writing." College English, 42 (1980):69-74.

Hartley, James (1999). "What Do We Know About Footnotes? Opinions and Data." Journal of Information Science, 3 (25):205-212.

Huckin, Thomas "Content analysis: What texts talk about." In C. Bazerman and P. Prior (eds.). What Writing Does and How it Does it, LEA: Mahwah, NJ: LEA, 2004. 13-32.

Huff, Sid. "Information Systems Maintenance." Business Quarterly 55 (1990): 30-32.

Hunt, Andy and Dave Thomas. "Software Archeology." IEEE Software, 19.2 (2002): 20-22.

Jansen, Frank, Van Lijf, Aimee, and Esther Toussaint. (2001). "A Note on the Evaluation of Footnotes and Other Devices for Background Information in Popular Scientific Texts." IEEE Transactions on Professional Communication, 3 (44): 195-201.

Joann. Observation. 15 June 2005.

Joann. Personal interview. 12 May 2005.

Joann. Personal interview. 12 August 2005.

Kajko-Mattsson, Mira. "A Survey of Documentation Practice Within Corrective Maintenance." Empirical Software Engineering, 10 (2005): 31-55.

Kohanski, Daniel. The Philosophical Programmer. New York: St. Martin's Press, 1998.

Kroll, Barry M. (1984). "Writing for Readers: Three Perspectives on Audience." CollegeComposition and Communication, 35.2 (1984):172-185.

Letovsky, Stan and Elliot Soloway. "Delocalized Plans and Program Comprehension." IEEE Software, 3 (1986): 41-48.

Lougher, Robert and Tom Rodden, (1993). "Supporting Long-term Collaboration in Software Maintenance." Proceedings of the Conference on Organizational Computing Systems, New York: ACM Press, 1993, 228-238.

Lowry, Paul B., Curtis, Aaron. and Michelle R. Lowry. (2004). "Building a Taxonomy and Nomenclature of Collaborative Writing to Improve Interdisciplinary Research and Practice." Journal of Business Communication, 41(2004): 66-99.

MacKinnon, James (1993). "Becoming a Rhetor: Developing Writing Ability in a Mature, Writing-Intensive Organization." In R. Spilka (ed.). Writing in the Workplace: New Research Perspectives, (pp. 41-55). SIU Press: Carbondale, Ill.

MacNealy, Mary Sue. Strategies for Empirical Research in Writing, Allyn and Bacon: Boston, 1999.

Miara, Richard J., Musselman, Joyce. A., Navarro, Juan A., and Ben Schneiderman. "Program Indentation and Comprehensibility." Communications of the ACM, 26.11 (1983): 861-867.

Miles, V.C., McCarthy, J.C., Dix, Alan J., Harrison, M.D., and A.F. Monk. "Reviewing Designs for a Synchronous-Asynchronous Group Editing Environment." In Computer Supported Collaborative Writing. Ed. Sharples, London: Springer-Verlag, 1993.

Norcio, Anthony F. "Indentation, Documentation and Programmer Comprehension." Proceedings of the 1982 Conference on Human Factors in Computing Systems, New York: ACM Press, 1982, 118-120.

Nurvitadhi, Eriko, Leung, Wing Wah, and Curtis Cook. "Do Class Comments Aid Java Program Understanding?" Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference, Washington DC: IEEE Computer Society, 2003, 13-17.

Oman, Paul W., and Curtis R. Cook. Typographic Style is More than Cosmetic.
Communications of the ACM, 33 (1990): 506-520.

Orr, Thomas (1999) "Genre in the Field of Computer Science and Computer
Engineering." IEEE Transactions on Professional Communication, 42.1 (1999): 32-37.

Pascal, Carrie L. "Enabling Chance Interaction through Instant Messaging." IEEE
Transactions in Professional Communication, 46.2 (2003):138-141.

Porter, Adam, Siy, Harvey, Toman, Carol, and Lawrence Votta. "An Experiment to
Assess the Cost-Benefits of Code Inspection in Large Scale Software Development."
IEEE Transactions on Software Engineering, 6 (1997): 326-349.

Prechelt, Lutz., Unger-Lamprecht, Barbara, Philippsen, Michael, and Tichy, Walter F.
(2002). Two controlled experiments assessing the usefulness of design pattern
documentation in program maintenance. IEEE transactions on software engineering, 28
(2002): 595-606.

Raskin, Jef. Comments are more important than code. ACM queue, 2 (2005). Retrieved
online, June 2005 from:

<http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=290>

Ray, Garry. "Disgruntled programmers needn't have the last word." PC Week, January 1989: 45.

Schneiderman, Ben, and Richard Mayer. Syntactic/semantic Interactions in Programmer Behavior: A Model and Experimental Results. International Journal of Computer and Information Sciences, 8 (1979): 219-238.

Sebesta, Robert W. Concepts of Programming Languages. 5th ed. Boston: Addison-Wesley, 2002.

Selzer, Jack. (1993). "Intertextuality and the Writing Process: An Overview." Writing in the Workplace: New Research Perspectives. Ed. Rachel Spilka. Carbondale, Ill: SIU Press, 1993. 171-180.

Sim, Susan Elliot, and Richard C. Holt. "The Ramp-up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize." Proceedings of the 1998 (20th) International Conference on Software Engineering, Washington DC: IEEE Computer Society, 1998, 361-370.

Simon. Observation. 3 June 2005.

Simon. Personal interview. 25 April 2005.

Simon. Personal interview. 10 August 2005.

Siy, Harvey, and Lawrence Votta. "Does the Modern Code Inspection Have Value?" Proceedings of the IEEE International Conference on Software Maintenance, Los Alamitos, CA: IEEE Computer Society, 2001, 281-289.

Soloway, Elliot, Pinto, Jeannine, Letovsky, Stan, Littman, David, and Robin Lampert. "Designing Documentation to Compensate for Delocalized Plans." Communications of the ACM, 31.11 (1988): 1259-1267.

Spinuzzi, Clay. "Compound Mediation in Software Development: Using Genre Ecologies to Study Textual Artifacts." In Writing Selves and Societies: Research From Activity Perspectives, Ed. Charles Bazerman and David R. Russell. Fort Collins, CO: The WAC Clearinghouse and Mind, Culture, and Activity, 2003, 97-124.

Spinuzzi, Clay. "Software Development as Mediated Activity: Applying Three Analytical Frameworks for Studying Compound Mediation." Proceedings of the 19th Annual International Conference on Computer Documentation, New York: ACM Press, 2001, 58-67.

Standish, Thomas A. "An Essay on Software Reuse." IEEE Transactions on Software Engineering, 10.5 (1984): 494-497.

Tenny, Ted. "Program Readability: Procedures Versus Comments." IEEE Transactions on Software Engineering, 14.9 (1988): 1271-1279.

Thomas, Richard A. "Using Comments to Aid Program Maintenance." Byte, May 1984: 416-422.

Winsor, D. (1996). "Writing Well, as a Form of Social Knowledge." In A.H. Duin, and C.j. Hansen (eds.). Nonacademic Writing: Social theory and Technology, (pp. 157-172). LEA: Mahwah, NJ.

Zokaites, David M. "Writing Understandable Code; Sure, You Know That You Should Comment Your Code, But How Well Are You Actually Doing It? Here's One Developer's Perspective on Documenting Your Application's Raison D'etre Through Comments, Naming Conventions and Clean Constructs." Software Development, 10.1 (2002): 48-50.