Compound Mediation in Software Development: Using Genre Ecologies to Study Textual Artifacts

Clay Spinuzzi

Division of Rhetoric and Composition
University of Texas at Austin
Austin, TX 78712
Clay.Spinuzzi@mail.utexas.edu
http://www.cwrl.utexas.edu/~spinuzzi

Abstract

Traditionally, technical communicators have seen the texts that they produce -- manuals, references, instructions -- as "bridging" or mediating between a worker and her tool. But field studies of workers indicate that the mediational relationship is much more complicated: Workers often draw simultaneously upon many different textual artifacts to mediate their work, including not only the official genres produced by technical communicators manuals but also ad hoc notes, comments, and improvisational drawings produced by the workers themselves. In this chapter, I theorize these instances of *compound mediatiation* by drawing on activity theory and genre theory. I describe an analytical framework, that of *genre ecologies*, that can be used to systematically investigate compound mediation within and across groups of workers. Unlike other analytical frameworks that have been used in studies of technology (such as distributed cognition's functional systems and contextual design's work models), the genre ecology framework highlights the interpretive and cultural-historical aspects of compound mediation that are so important in understanding the use of textual artifacts. The analytical framework is illustrated by an observational study of how 22 software developers in a global corporation used various textual artifacts to mediate their software development work.

Among those who study workplace communication, technical communicators are particularly interested in how textual artifacts help to mediate work. Indeed, to a great extent technical communication involves developing texts to mediate between workers and their tools – for instance, manuals that explain how workers can use a particular tool to perform given tasks (Paradis, 1991). Traditionally this has been taken to be a relatively simple (albeit often difficult) task: One determines what sorts of tasks a worker performs, then describes those tasks in such a way that the worker can understand. The mediational relationship is conceived as a "bridge" between the worker and her tool.

But field studies of workplace technical communication have recently complicated the picture. In practice, workers appear to make use of many diverse textual artifacts in complex, coordinated, contingent ways to get their work done. These texts range from official to unofficial (Spinuzzi, 2002b), specialized to mundane (Johnson, 1998; Winsor, 2001), and are often pressed into service in unpredictable, idiosyncratic ways (Henderson, 1996; Johnson-Eilola, 2001). In particular, studies based on user-centered design methods have helped technical communicators to examine how textual artifacts co-mediate work (Johnson, 1998; Raven & Flanders, 1996).

Published February 1, 2003

In this chapter, I use the term *compound mediation* to refer to the ways that people habitually coordinate sets of artifacts to mediate or carry out their activities. For instance, in the activity of software development, a developer might simultaneously use software manuals, existing code, online language references, and scratch paper as she writes new code. Each artifact – and each type of artifact – helps to shape the activity and enable people to perform their many actions. In fact, one can argue that artifacts become useful to a given set of workers only through the ways in which those workers intermediate or juxtapose other artifacts (see Hutchins, 1995a; Spinuzzi, 1999).

Although technical communicators are fundamentally concerned with compound mediation (for instance, documentation mediates between a reader and tools), few technical communication researchers consistently use analytical frameworks for examining compound mediation. That does not mean that such studies are poor – many are excellent studies (Haas, 1996; Henderson, 1996; Mirel, 1988; Mirel, Feinberg, & Allmendinger, 1991) – but since they do not apply analytical frameworks for studying compound mediation, they miss chances to systematically compare mediatory systems, improve an analytical framework, or scale up to examine larger numbers of artifacts.

Recently, a variety of frameworks have been developed for exploring compound mediation, such as contextual design's *work models* (Beyer & Holtzblatt, 1998; Coble, Maffitt, Orland, & Kahn, 1996; Page, 1996) and distributed cognition's *functional systems* (Ackerman & Halverson, 1998, 2000; Hutchins, 1995a, 1995b; Rogers & Ellis, 1994). Those who research workplace communication, particularly those coming from backgrounds in rhetoric and technical communication as I do, find such frameworks to be valuable for at least three reasons. First, since such analytical frameworks have been formalized, they are easier to approach systematically. They provide common concepts and terms. Consequently, they can be standardized, giving researchers criteria for what to study. Thus an analysis based on an analytical framework can be held up to standards of reliability and validity. Second, once articulated, frameworks can be examined and incrementally improved by various researchers. They become objects of study themselves (as in this paper) and therefore can be studied more critically. Consequently, later researchers might improve on the reliability, validity, and analytical power of existing frameworks. Third, since analytical frameworks are developed to offer a systematic analysis, they tend to be scalable. For instance, work models can be applied to very small numbers of artifacts (say, in an office cubicle) or very large numbers (say, across an entire company). Yet, as I've argued elsewhere (Spinuzzi, 2001b), these frameworks tend to focus on managerial or computational perspectives rather than the interpretive issues that centrally concern rhetoricians and technical communicators.

One analytical framework that does address these interpretive issues, the *genre ecology*, is currently in the process of being developed and formalized. Based on genre theory and activity theory, the genre ecology framework highlights the interpretive and cultural-historical aspects of compound mediation. In this paper, I illustrate this framework by using it to analyze data I collected during an informal[1] field study of software developers. Finally, I discuss possible future development opportunities for this framework.

The Genre Ecology Framework

The genre ecology framework was developed by technical communicators specifically for describing and investigating compound mediation (Spinuzzi, 2001b, 2002b; Spinuzzi & Zachry, 2000; Zachry, 1999). Genre ecologies have roots in earlier frameworks such as genre sets (Devitt, 1991) and genre systems (Bazerman, 1994), but they also draw from information ecologies (Nardi & O'Day, 1999; Zachry, 2000) and distributed cognition's tool ecologies (Hutchins, 1995a; see also Freedman & Smart, 1997). What sets genre ecologies apart is the focus on contingency, decentralization, and stability (Spinuzzi & Zachry, 2000) as these dynamic ecologies gain, adapt, and discard genres.

Like frameworks such as contextual design's work models and distributed cognition's functional systems genre ecologies provide a descriptive model of compound mediation. But whereas work models provide an agent-centric managerial view and functional systems provide an artifact- or system-centered computational view, genre ecologies provide a community-centered interpretive view. Genre ecologies highlight idiosyncratic, divergent understandings and uses of artifacts and the practices that surround them as they develop within a given cultural-historical milieu.

In the next section, I briefly go over the genre ecology framework's pedigree in activity theory and genre theory.

Activity Theory and Compound Mediation

Activity theory is a cultural-historical psychology concerned with labor activity. In this account of how people coordinate and enact cyclical, objective-driven activities, activity is conceptualized as inherently social and intricately bound up with joint (community) labor (Leont'ev, 1978; Leontyev, 1981; Engeström, 1990, 1992; Nardi, 1996a).

A central concept in activity theory is the *activity system*. Activity theory posits that in every sphere of activity, one or more *collaborators* use *artifacts* (including physical and psychological tools) to transform a particular *objective* with a particular outcome in mind (Figure 1). For instance, a software developer (a *collaborator*) may use various artifacts to transform an existing software program into a newer program (the cyclical *objective*). She does so to achieve various outcomes.

Yet the developer's activity does not spring from herself alone. It is intimately related to her *communities*, including the company where she works and the larger community of her profession; her relationships to these communities are mediated by *domain knowledge*, including the habits she has developed for using various artifacts, work regulations, and ethical guidelines. Similarly, the relationship between the community and the object is mediated by the *division of labor* within that community: software developers write code, often helped by managers, software documentors, and others in various ways. So the developer's job is intimately tied to the cultural-historical milieu in which it is performed.

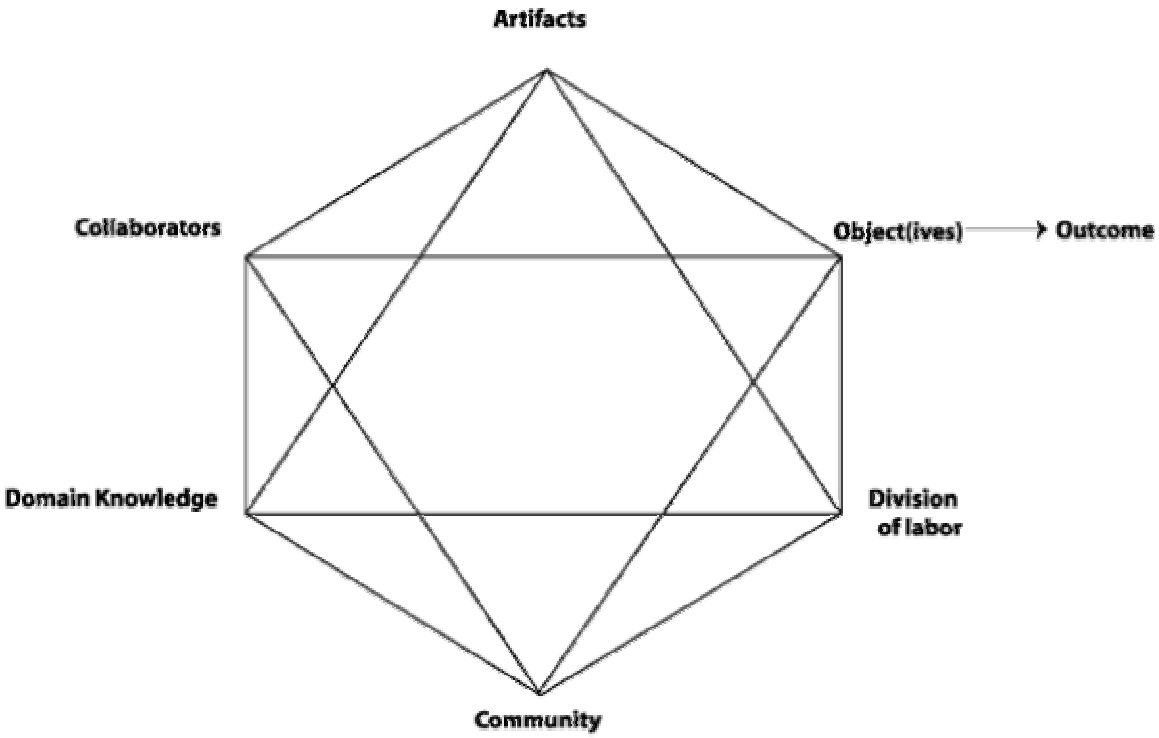

Figure 1. An activity system, in which collaborators (human actors or organizations) engage in mediated relationships to transform their objectives.

Although Figure 1 may seem to imply that an activity is mediated by a single artifact, activity theorists have begun to explore how *compound* mediational means can collectively mediate activities. Susanne Bødker's work in human-computer interaction, for instance, illustrates how artifacts relate to one another, being situated in a "web of artifacts" (1996, p.161; see also 1997) that jointly mediate activity. Others using variants of cultural-historical theory have advanced similar concepts of compound mediation, using different terminology (Bazerman, 1994; Cole, 1996; Freedman and Smart, 1997; Hutchins, 1995a; Orlikowski and Yates, 1994).

Genre Ecologies

To discuss how such groups of artifacts mediate activities, I turn to Edwin Hutchins' "ecologies of tools." Hutchins found that the many artifacts used on a naval vessel (such as astrolabes, compasses, sextants, and calculators) are arrayed and employed by multiple workers to transform data. He sees these transformations as part of a computation in which "each tool creates the environment of the others" (p.114). There tools are connected in multiple, complex, and often nonsequential ways. Furthermore, they co-evolve: changes in one lead to changes in others. For instance, functions sometimes move around from astrolabe to quadrant to cross staff to sextant (p.113-114); such movement is made possible by the tools' interconnections in the ecology. The ecology itself — not its individual tools — is the mediator of the activity.

The genre ecology framework is based on this notion of tool ecologies. But rather than focusing on the functional aspects of tools, it focuses on the interpretive aspects of *genres* of tools, i.e., artifact types that are cyclically developed and interpreted in ongoing activities. In any given activity system, artifacts become familiar to workers over time, so much so that workers begin to interpret artifacts as instantiations of genres. These genres of artifacts collectively mediate the workers' activities, and in doing so they become interconnected with each other in mediational relationships. Such interconnected genres can be considered *genre ecologies* (Spinuzzi, 2002b; Spinuzzi & Zachry, 2000; Zachry, 1999).

To illustrate how the genre ecology framework can be applied to workplace studies to examine complex issues of interpretation, next I analyze the results of an informal workplace study in terms of genre ecologies.

Overview of the Study

I conducted this study at Schlumberger Oilfield Services as a member of the Usability Services for Engineering Research (USER) team. The study, conducted over 10 weeks during the summer of 1997, involved observations of and interviews with 20 software developers and interviews only with two additional developers.[2] I collected these observations and interviews to inform the design of an information system meant to provide quick access to information on library routines and datatypes, the "building blocks" of Schlumberger's programming code.

Background: Questioning the Usability of Code Libraries

Schlumberger is a multinational corporation that provides information tools for the oil industry. In 1997, Schlumberger's oilfield software consisted of more than 30 million lines of code dedicated to analyzing and interpreting the seismic and drilling data that were collected by Schlumberger equipment at drilling sites around the world. This code is developed in-house, and software developers continue to generate more code as they maintain and develop products. Schlumberger encourages developers to reuse existing chunks of code from the corporation's code libraries — collections of commonly useful datatypes and routines (explained below) that are accessible to all developers working on a given project. Yet developers sometimes have a difficult time finding appropriate code in the libraries. Consequently, they "reinvent the wheel," writing new code that is sometimes not as efficient or well-tested as the existing code in the libraries. Since the code libraries can change daily, printed reference guides and even static online guides quickly become out of date. In-house quantitative studies on code-sharing suggested that developers are often not availing themselves of the routines in the code libraries (McLellan, Roesler, Fei, Chandran, & Spinuzzi, 1998).

Schlumberger's library code contains two related types of code: datatypes and routines. *Datatypes* are units for storing information; these units can contain complex combinations of integers, floating-point numbers, characters, and other kinds of data. These datatypes are manipulated through *routines*: chunks of instructions that involve transforming the data stored in

datatypes. Developers can use a library's datatypes and routines within a given code library in any program that explicitly refers to that library.

In the study, I concentrated on the artifacts that developers used as they comprehended (interpreted) and produced (wrote) code. As I argue below, the developers' comprehension and production of code was mediated by the ecologies of genres that surrounded their work.

The Study: Investigating Software Development

In the study, I investigated how developers used tools as they wrote code, with a particular interest in how the USER team could modify or introduce tools to better support program comprehension and production. To better examine how the developers interpreted (comprehended) and used these tools in their activities, I turned to activity theory and genre theory for a suitable framework. I originally conducted this study with the following research questions:

- How are developers currently finding appropriate library code to use? That is, how do developers find out about the datatypes and routines that currently exist in Schlumberger's libraries?
- How do the developers use artifacts (such as search tools, features of the code, communication media, and manuals) when comprehending and producing code?

These questions led me to examine how groups of artifacts jointly mediate activities. To answer these questions, I used the analytical framework of the genre ecology. Below, I discuss the research sites I visited, the participants I observed and interviewed, and the methods I used as I investigated the sites.

Research Sites

Schlumberger consists of a number of related large business groups. I visited three of these groups (hereafter sites Alpha, Bravo, and Charlie) during the course of my research. These groups used similar, but different, interrelated genre ecologies. (In a closer analysis, one might study each site as a separate activity system that shares strong commonalities with the others because they all engage in the more general activity of Schlumberger software development.) These three business groups, originally separate organizations, were acquired by Schlumberger at different times during the last decade. Much of their code is "legacy code" that dates from that pre-Schlumberger time.

At each site, I observed and interviewed a group of software developers roughly proportional in experience and gender makeup to the total population of the site. Interview questions and observations focused on the artifacts that they used as they produced and comprehended code, particularly in how they characterized their interaction with these artifacts.

Site Alpha maintained a compilation of seismic processing systems that had been unified into a seismic data interpretation system. This system was mainly written in FORTRAN, although some new additions were being written in C. Developers worked on a UNIX platform. I observed two developers at work and interviewed two others. These developers, like most at Site Alpha, were oriented towards internal users: that is, they wrote code that was primarily used by Schlumberger employees and had contact primarily with Schlumberger engineers.

Site Bravo maintained a variety of data management products used to interpret oilfield data. This system was mainly written in C and C++, although two of the developers I interviewed were porting (i.e., translating) an existing product to Java. Like the developers at Site Alpha, these developers worked on a UNIX platform. I observed eleven developers at work here. These developers, like most at Site Bravo, were more oriented to outside customers: they tended to review the system's user manuals to find out how their code was "supposed" to work (from the users' standpoint) and spent more time on user-oriented details such as the user interface.

Participants

The 22 participants drawn from the three sites constituted a complex division of labor, partly defined by the phases in the production cycle (development, maintenance/debugging, and documentation). The interviewed developers had a wide variety of experience. Nevertheless, developers tended to have strong commonalities within and across the three sites. As I argue below, the developers' work was mediated by the genre ecologies at the three sites as they attempted to find suitable code and to comprehend and produce code.

Site Charlie maintained products for extracting, processing, and interpreting data on location (i.e., as it is collected at the drilling site). This system was written primarily in C and C++. Unlike the other two locations, Site Charlie was a "Windows shop": Developers worked on Microsoft Windows NT workstations rather than UNIX workstations. I observed seven developers at work here. These developers, like those at Site Alpha, were oriented toward Schlumberger employees who used their software in the fields.

This study was approved by a human subjects committee before research began.

Methods

For 20 of the participants, I used the following methods:

Opening interview. In audiotaped interviews, I asked each developer a series of questions meant to explore his or her background with software development, Schlumberger, and the code with which they were working (see appendix).

Observation. After completing the opening interview, I silently observed the developer as he or she worked, taking field notes. I particularly focused on artifacts that developers used as they attempted to find and use existing code. Observations lasted 30 minutes to an hour, but averaged

about 45 minutes. The observations allowed me to compile a list of some of the artifacts used in the participant's labor and the domain knowledge associated with them. The observations also gave me insight into the object of the labor (the code), the community, and the division of labor.

Closing interview. At the end of the observation period, I conducted a stimulated recall interview in which I asked the developer about (a) the artifacts that I had seen him or her using and (b) artifacts that he or she did not use in that session. Interviews were audiotaped. The closing interview allowed me to round out the list of artifacts, as well as discuss the other parts of the activity system in which each participant labored.

I was not able to observe the other two developers at work, but did conduct the opening interview and also asked them about artifacts they had used when finding information. After conducting the interviews and observations, I (a) categorized each developer's work as developing, maintaining/debugging, or documenting; (b) listed the artifacts I had observed them using and categorized the artifacts within genre categories; and (c) listed questions that developers used each artifact to answer, based on the interview responses.

Results and Analysis: Mediation Within and Across Genre Ecologies

Using the data from the interviews and observations, I constructed Figure 2, which shows the overall activity system[3] of the developers. Bear in mind that this figure describes a general activity system that extends across sites. In the following analysis I explore the more specific activity systems of the individual sites, drawing out some of their differences. Of the most interest, though, is the abundance of genres in the ecology and their complex interconnections.

As Figure 2 shows, the developers operated within a complex activity system with a variety of artifacts, domain knowledge governing the use of those means, and a complicated division of labor. In the analysis below, I focus on these three elements. But the different sites were also activity systems in their own right: Their activities differed significantly in artifacts, domain knowledge, and objectives. Some of these differences are mentioned in the brief descriptions of each site above; others are drawn out in the analysis.

The genres in Figure 2 have complex relations with each other, forming an ecology of genres that developers employ to find specific types of information about the code that is the object of their work. As Figure 3 depicts, these genres are connected – and jointly mediate the activity – in a variety of different and complex ways.

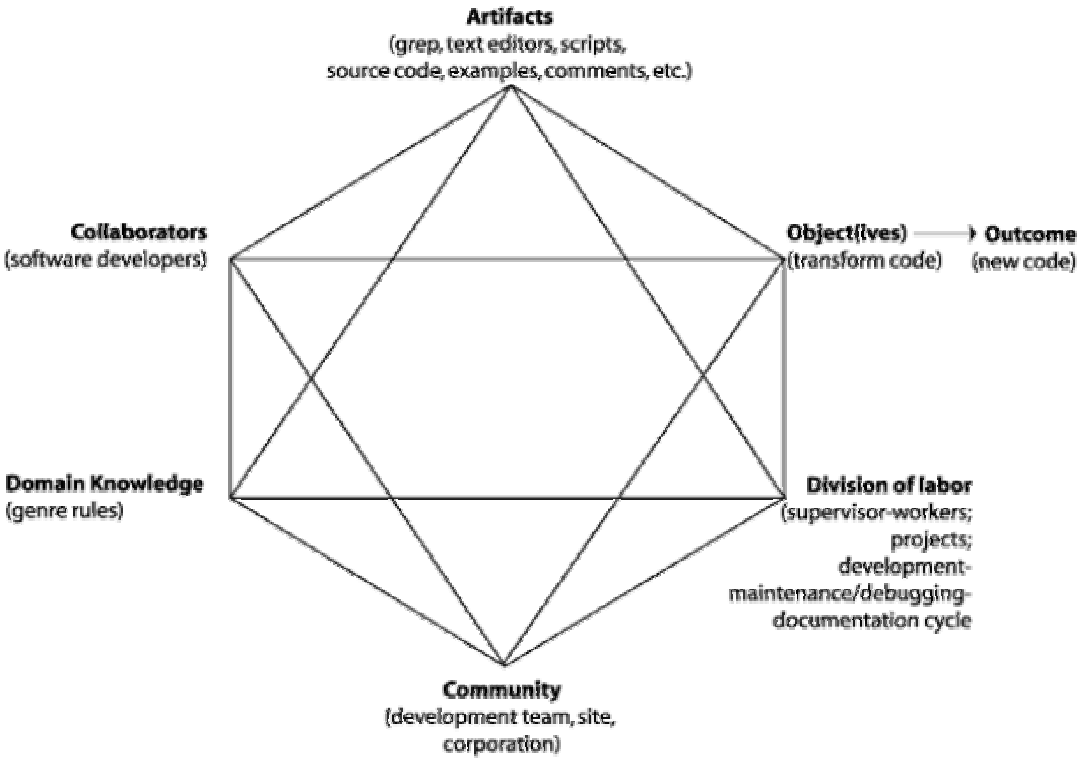

Figure 2. The software developers' general activity system. Activity systems could be depicted for each site as well.

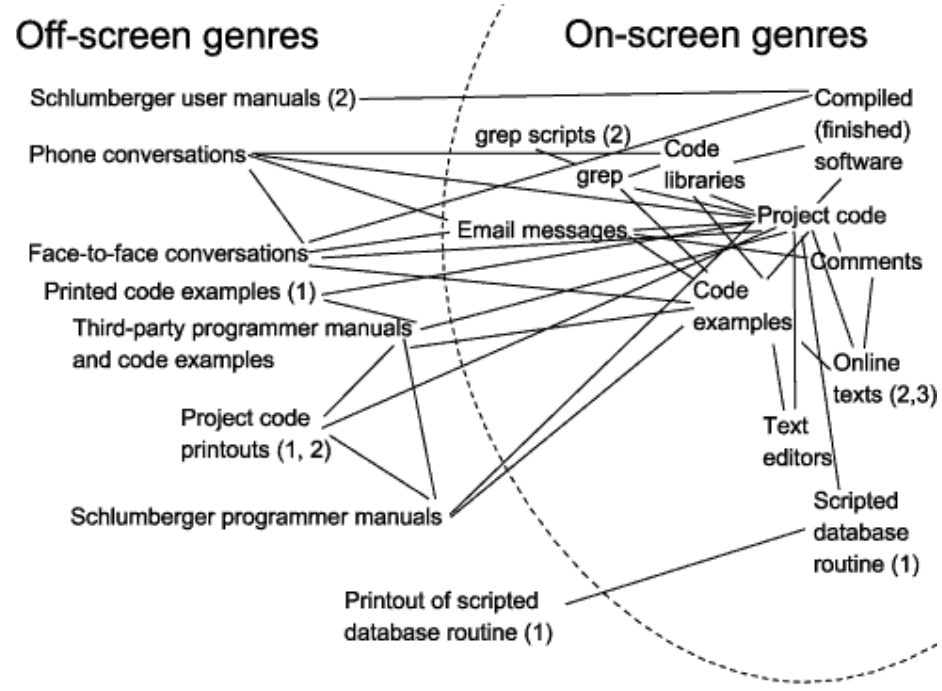

Figure 3. The genre ecology of the developers' activity system. Numbers in parentheses mean that the indicated genres are only used in the ecologies of specific sites: (1) corresponds to Site Alpha, etc. Lines indicate mediational relationships among genres.

Genres within the dotted circle are on-screen genres (i.e., they exist on the computer screen).

At all three sites, the elements of the activity system (Figure 2) were substantially the same. The artifacts, for the most part, were materially the same, available across all sites, and organized in generally the same ways. The object of the activity (the code) was more or less the same — in fact, much of it was shared among sites. The division of labor was at least formally the same across sites. Yet within these different sites (communities), the domain knowledge (including the unwritten rules, habits, and practices governing artifacts) was sometimes quite different. Consequently, developers sometimes perceived artifact types or genres as having different uses, possibilities, and levels of importance at different sites. These genres were constrained by the characteristics of the genres and the code, as well as the historical development of practices and the interrelationships of genres at the various sites.

Genre Ecologies at the Three Sites

Figure 3 shows the genre ecologies at the three sites. The three ecologies were similar — thus the single diagram. Numbers in parentheses mean that the indicated genres were only used in the ecologies of the enumerated sites. For instance, developers used grep (a software tool used to find strings of characters in text files) at all sites, but they used grep scripts only at Site Bravo.

The lines indicate mediational relationships among genres. That is, if two genres are connected by a line, I observed those two genres being used in conjunction: one mediated actions performed on the other. For instance, developers at all sites interpreted project code by consulting third-party programmer manuals, code examples, and comments, so in Figure 3 these genres are all linked to the project code by lines. On the other hand, developers did not use Schlumberger's user manuals to help them interpret comments, so these genres are not linked.

Figure 3 gives us a partial idea of what genre ecologies looked like at the three sites. However, it is just a partial idea; a more extended study might turn up dozens or hundreds of genres and might make it practical to analyze subgenres (such as subgenres of online texts). In addition, the diagram gives us an informal method of identifying genres that were densely connected. (For instance, the object of the developers' work was the project code, which in the diagram is connected to nearly every other genre in the ecology. On the other hand, Schlumberger user manuals were used for only one purpose, to understand the finished software.)

Below, I compare some of these genres between ecologies and within ecologies. I start by showing how individual artifact types can be considered genres. Then I show how these and other genres interact to jointly mediate activities.

The Generic Nature of Artifacts: Inter-Ecology Comparisons

In this analysis I discuss how developers at the three sites, although using the same materials, used them in quite different ways. I analyze these artifacts in terms of genre.

Figure 3 depicts the three sites' genre ecologies "overlapped" as it were. Most of the genres were available at all three sites and connected to other genres in similar ways. But the exceptions were telling ones: They suggested that similar genres were perceived as having different affordances, different uses, and different degrees of usefulness at the three sites, partially because of how these genres interacted with other genres in the ecologies. Below, I discuss two examples — the grep utility and comments embedded in the code — and use them to demonstrate how different artifacts interact in mediating the activity system.

Grep and Grep Scripts: Interpreting and Using a UNIX Utility

The grep utility is a program that searches for specified strings of text within sets of files. For instance, to find the name "reversestr" in the file "somefile.c" which resides in a given part of the system, a software developer might type this command at the system prompt:

```
grep "reversestr" /home/jones/project/fileio/somefile.c
```

This command is terribly powerful, since it allows developers to quickly identify points at which a given routine or variable is being used. But at the same time, it takes considerable time to formulate and type such a command. In response, some developers at Site Bravo had augmented grep's capabilities by assembling *scripts*, or collections of commands that invoke grep for specific files in specific directories: A developer who tired of repeatedly entering each command in Figure 4, for instance, might collect them into a single file and run them all at once by typing the file's name at the command line.[4] So, for instance, a developer might assemble a script of grep commands for a certain project and call the script "pgrep." Then, to search the appropriate files for a certain string of letters — say, the routine name "reversestr" — the developer could type *pgrep reversestr* at the command line, and grep would search each file in turn.

```
grep $1 /home/jones/project/fileio/*.c

grep $1 /home/jones/project/screenio/*.c

grep $1 /home/jones/project/fileio/*.h

grep $1 /home/jones/project/screenio/*.h
```

Figure 4. A (fictional) grep script similar to those used by developers at Site Bravo. In each invocation, $1 is automatically replaced with the string to be found.

Six developers were observed using grep; in the interviews, 12 discussed using grep for finding information in the libraries.

The grep utility was used repeatedly and cyclically by developers to accomplish repeated actions within their activities. And the repeated use gave rise to stabilized-for-now rules (habits) for operating it. These rules were not simply embedded in the software: Developers used grep with strikingly different rules at the three locations. Rather, the rules were demonstrably affected by other available genres, the search environment, and the community's domain knowledge. Grep mediated activities in different ways at the different sites, even though the functional qualities of grep were the same at all three sites.

Grep use at Site Bravo. For instance, developers at Site Bravo tended to create grep scripts that searched specific files stored in specific paths[5] related to the developer's current project. These scripts were not simply passed from one developer to another — as one developer told me, each developer knew enough about grep and various scripting languages to create these scripts for him- or herself, and in any event each developer worked with different paths, so the scripts had to be different for each developer. And, the developer pointed out, a grep script was an obvious solution to the limitations of grep, something that (he assumed) would occur to any competent software developer. Yet this "obvious solution," although widespread at Site Bravo, was unheard-of at the other two sites. Although developers at each site had access to the same tools (grep and some scripting language), grep was a qualitatively different artifact at the three sites; it had different uses and meaning because the practice of scripting had only developed at Site Bravo. (Unfortunately, this 10-week study did not allow me enough time to collect historical data that might suggest why the sites developed different rules.)

To illustrate the qualitative differences of grep across sites, I contrast Site Bravo's practice of scripting with that of information searching at Site Charlie and Site Alpha.

Grep use at Site Charlie. Site Charlie's developers used grep extensively, but only for searching files in a particular directory. Developers conducted searches across paths using a separate tool, a Windows-based text editor. Many of Site Charlie's developers preferred grep because it is faster than the text editor, but none had assembled grep scripts — although those scripts, like individual invocations of grep, would theoretically be faster than the text editor's search feature. When asked, developers at Site Charlie indicated that the idea of grep scripts simply hadn't occurred to them — and that they probably would not write grep scripts in the future. The initial, intensive labor involved in writing the script seemed like too much trouble compared with the less intensive but repeated labor of using the text editor.

Grep, then, was interpreted and used differently at Sites Bravo and Charlie. At Site Bravo, developers automated repeated tasks such as searching multiple baselines. This automation required initial investment but minimal interaction each time the baselines were searched. That is, the Site Bravo developers were willing to devote substantial time to devising grep scripts, believing that the initial effort would pay off in easier searches. Site Bravo's developers had introduced a new genre into the ecology, that of grep scripts, and in consequence grep had become more useful to them.

At Site Charlie, on the other hand, developers made little initial investment and engaged in moderate to heavy interaction each time the baselines were searched. Site Charlie developers devoted no time to devising scripts; rather, they used a text editor, a tool that required considerably more interaction each time it was used to search for strings. Rather than introducing another genre into the ecology, Site Charlie's developers distributed the task of searching between grep and another existing genre, the text editor.

Grep use at Site Alpha. At Site Alpha, developers used grep to search specific directories, but did not develop grep scripts. In fact, practices of baseline-wide searching did not become apparent in either the observations or the interviews, except in one case: one developer did demonstrate an analogous script for searching a database of code, using database commands rather than grep scripts. This script allowed him to find the desired code within the code archives rather than actual paths. The database script was simple, but this developer had to type it each time, since he had not created a permanent electronic version. To aid him, he printed the script and taped it to the desk beside his keyboard.

At Site Alpha, then, this developer conceived of searches in a third way. Searches entailed moderate initial investment (typing out and printing a script) and moderate interaction (using the script to guide the developer's own actions each time). By introducing a new genre into the ecology, the printed script, this Site Alpha developer had found yet another way to mediate his searches.

The data above suggest two things. One is that, to paraphrase Hutchins, genres embody distributed cognition. That is, a genre — such as, say, a grep script — is a material solution to a problem once faced by its originator, a solution that was successful enough to be used repeatedly by others.[6] When workers apply a genre to a problem, in a sense the problem is being partially solved by those who developed the genre. Over time, the genre becomes familiar enough to its users that it is perceived as obvious (and in some cases trivial).[7] That effect is difficult to see when one is embedded in the community using the genre. The developers at Site Bravo, for instance, were familiar enough with grep scripts to consider them an "obvious" solution, not an innovative answer to a difficult problem. At the same time, rules for use are not "designed into" these artifacts. At the three sites, many of the material tools did not vary, but the domain knowledge did, and that variation resulted in artifacts that were qualitatively different, that is, perceived as having quite different uses and possibilities.

The second thing the data suggest is that interactions among genres can influence genre use. For instance, all three sites faced the same problem: how to find code across multiple paths. Although all three had the same basic tools (grep, text editors, databases), each site adapted a different tool to occupy this ecological niche. In doing so, they made compromises in terms of initial investment and effort per use.

Comments: Interpreting and Using Notes Embedded in the Code

Comments are non-program text that developers embed in both library and project code. (In this study, developers only used comments embedded in the project code.) Comments do not have

any effect on how the computer runs the program. They are generally assumed to be notes that developers include to help later developers interpret the code (see Takang, Grubb, and Macredie, 1996; Tenney, 1988), although in this research I found many comments that had other uses. Figure 5 shows C-style comments embedded in the code between the character combinations `/*` and `*/`. When asked about how they comprehend routines and datatypes in the libraries, eleven developers mentioned looking at the comments to aid comprehension — although, as I explain below, developers at the three sites used comments in quite different ways.

Comment use, like grep use, differed by site. For instance, at Site Alpha and Site Bravo, developers tended to read, write, and maintain multiline comments as a rule. Developers used these comments as notes for interpreting code, but also to signal plans for maintenance, as in the comment shown in Figure 5. (The product name has been changed to protect confidentiality.)

```
/* Product A is using a GDM attribute to indicate the deviation status */

/* of a well even though they are using the GDM8 model. To improve the */

/* the communication of this information to Product A we are adding this */

/* GDM9 attribute to the GDM8 model as a private extension. But, we */

/* only do this if we are still using GDM8. Once we switch to GDM9, */

/* an error message will be issued. */
```

Figure 5. Multiline comment used by Site Bravo developers to coordinate work.

Such comments were common in the Site Alpha and Site Bravo code. They provided important resources for interpreting the code in relation to future changes as well as coordinating work. Developers at these sites often remarked on the utility of these comments both for sharing information with others and for reminding themselves of changes. These developers were scrupulous about documenting code with comments.

On the other hand, developers at Site Charlie were far less likely to write or read multiline comments. At this site, developers had continually maintained and updated legacy code dating from the early 1980s, but they had not updated comments dating from that time, subverting the comments' usability as an interpretive tool. One developer explained that he found comments to be useless:

> **P16**: Well, people put in comments and make changes and they never keep up with the comments. ... That's about like if you're gonna drive down the road, and you're wantin' to get out on the street, and you see someone comin' down the road, they've got their blinker on, okay, are you going to trust that that car is going to turn?

The attitude expressed by this participant was pervasive throughout Site Charlie. Rather than being helpful aids to interpretation, comments were perceived as actually misleading. Some

developers actually regarded multiline comments as *evidence of poor coding* rather than tools for interpreting code or coordinating work:

> **CS:** ... you said that ... you try and make [your code] very clear for the other people who are going to be maintaining it after you. Do you put in a lot of inline comments in your code?
>
> **P18:** If it's necessary. Not too many comments. Because, uh, to me, if you look at the program that has lots of comments, it means the program wasn't structured right. If you have meaningful variable names and meaningful data structures set up, organization in your program, you don't need to have that much comments. You need to have enough comment not to clutter the code. Because you can have so much comment to confuse the user. You know, have more comments than code.

At Site Charlie, criteria for "good code" — interpretable and structured code — went far beyond explicit criteria such as robustness and efficiency. This low regard for comments at Site Charlie gave rise to interesting uses of artifacts, uses that were not found at other sites. For instance, one developer set his text editor to change the color of comments in the code he maintained – visually eliminating them from the code.

> **P20:** But [the editor's feature of displaying text in different colors is] nice especially in things like the comments. Now for me, it, it's probably not, useful in the way that it was intended, see I use it to almost get rid of the comments, I make them a light grey color so they don't stand out, so it's easy to glance and see where the code is, and not be distracted trying to, you know, with something that's a comment.

Another developer used comments primarily as landmarks for navigating through the code, rarely if ever reading them to understand the code's workings:

> **P21:** Well, when it's your own code, and you see it every day for hours, you're not reading comments anymore. ... But I use them for locating my code, really. If I have 50 lines of code without a comment I get lost. It takes me a while to actually read the code and find out what it's doing. But if I have comments I can separate it into sections, and if I know it's the second section in the function, I can go right to it. ... They're just markers for sections of code.

Like grep, then, comments were qualitatively different genres at the three sites — not because of inherent characteristics of the comments themselves or because of a different function of the work at each site, but because the production, comprehension, and use of comments depended on their history within the different activity systems and with the different genre ecologies that mediate them. The genres developed to support different kinds of work, that is, to support different sorts of usability by interconnecting with different interpretive habits at the three sites. At two of the sites, comments had been historically valued and maintained, and consequently developers saw the value in reading, believing, and producing them. Comments had become central to some of the actions carried out at those sites. But at Site Charlie comments had been

historically disregarded, and consequently developers found ways not to read them, or to read them in entirely different ways, as placemarkers and as indicators of code quality.

Furthermore, these developers at Site Charlie avoided maintaining and producing multiline comments, ensuring that future developers at this site would continue not to use such comments to comprehend or produce code. So comments at Site Charlie constituted a quite different genre from those at Sites 1 and 2. What is a useful comment? One that provides resources for interpreting and maintaining code? Or one that provides code separations and allows developers to evaluate code quality? The answer depends more on one's activity system than on the structural qualities of the comment itself.

Up to now I've selected single genres and shown how they mediated work differently in different activity systems. At this point, I turn my attention to intra-ecological comparisons, that is, how different genres in the same ecology relate to each other.

Ecological Relations Among Genres: Intra-Ecology Comparisons

Much research into computer program production and comprehension suggests that programmers interpret and use their programs in a way that can be explained in terms of genre. As I've discussed elsewhere (Spinuzzi, 2002a), over the last two decades several theoretical models have been developed that view program comprehension and production as an interrelated set of habits and hypotheses that programmers form, not to communicate with the computer itself, but to signal their logic and intentions to other programmers. (For instance, I quoted P18 earlier as saying that "if you look at the program that has lots of comments, it means the program wasn't structured right"; even though a program might run on a computer, it may not meet the criteria of "good code" because it is not impressive, elegant, or usable for other programmers.)

These models of programming tend to use terms such as "schema" (Rist, 1989, 1991), "beacons" (Brooks, 1983; Wiedenbeck, 1986, 1991), and "plans" (Soloway and Ehrlich, 1984; Boehm-Davis, Fox, and Philips, 1996; Pennington, Lee, and Reder, 1995).[8] They tend to portray programmers as sectioning code into interpretive rather than structural units and examining or producing those units cyclically rather than linearly. Although this is not the place to go into a detailed analysis of such models, they describe programming as an activity that involves the development and refining of stabilized-for-now interpretive strategies for understanding code — that is, strategies associated with genre (Schryer, 1993; cf. Bakhtin, 1986; Bazerman, 1994; Russell, 1997).

Thinking of code as a set of genres in an ecology of work activity helps us to understand a common activity of software developers: examining one piece of code to understand another. The project code becomes an indirect mode of communication for the developers, a sort of narrative that developers can use to find useful library code and understand how it is used.

Using code examples with libraries. Looking at the library code allowed developers at all three sites to learn certain (abstract, definitional) things about the routines and datatypes — for instance, the arguments to be passed to the routines, the makeup of the datatypes, and the specifics of inner workings such as memory usage. These were all data that developers tend to use when they (a) knew the exact name of the routine or datatype to examine, (b) were just learning the inputs of the routines, or (c) needed to know specific information about the inner workings of the code.

But library code gave little information about the uses to which the routines and datatypes could be put. And workers at all three sites found that they needed contextualization of the code far more often than they needed specific information:

> **P14:** Yeah. Examples are important because basically they're, they're kind of a parallel proof of concept. Or they show what the boundary conditions, what you can do and can't do. Because just the definitions are not appropriate alone. When you want a certain function to do something, definitions are great. If you're scanning through this here, you want to know the constraints of it. But if you want to know its usage, the usage scenarios, you gotta have examples.

Here, P14 discusses using code examples — concrete examples in living code as well as abstract examples in a book — in conjunction with other genres such as code libraries and programmers' references (which provide the dead specimins, the "definitions" and "constraints" of functions). The various genres are different resources with different roles in the ecology. Developers used each genre to co-mediate the others, bringing the ecology's genres to bear in different combinations on the object, the code.

Using code examples with search genres. In addition, looking at project code provided developers with a sort of reverse search mechanism. The project code was co-mediated by a variety of search genres, providing a flexible set of configurations for carrying out various kinds of searches. Recall that if developers were to use search genres such as grep scripts, text editor commands, or database commands to locate code, they needed to know the exact name of a routine or datatype when searching for it in the library code. But if developers did not know the exact name — a situation that happened more often than not — they tended to look in the project code for an example, a stretch of code that did something similar to the task they wanted to accomplish. They found examples of procedures in the code, procedures that included calls to library functions that they found useful in their own code. Code examples complemented the other search genres: Rather than looking up a function to find scenarios in which it was used, developers looked up scenarios to find functions they might use.

The labor of finding information was complex. Developers sometimes had to work both ways, from examples to formal definitions and from formal definitions to examples. On one hand, an interesting routine in example code might lead the developer to search the project code and libraries for a formal definition of the routine. On the other hand, a formal definition in a code library, manual, or technical specifications might lead the developer to search the project code

for examples. If only one of these genres was available, it would be less usable, because developers used the genres together – as part of an ecology – to comprehend the code.

In fact, part of the reason for Schlumberger's "code bloat" or expansion of redundant code (McLellan, Roesler, Fei, Chandran, & Spinuzzi, 1998) was that this co-mediation was not always sustained. Developers often relied on examples that did not always fit their tasks, and consequently they did not find the more appropriate routines and datatypes. Furthermore, developers sometimes used these examples to avoid writing code: They would sometimes copy an example, paste it in the spot where they wanted new code, then modify the example to meet their needs. Such code would sometimes contain references to routines and datatypes in the code libraries. The developers did not necessarily need to know what routines and datatypes originated in the library; in fact, they often seemed to be unclear as to exactly where some routines and datatypes originated. One developer at Site Charlie, for instance, who had been on the job for only a few months, revealed to me that he did not always understand the code he produced. Rather, he programmed as a bricoleur: He found similar stretches of code, pasted them into the proper spot, and tweaked them until they worked. This nonhierarchical programming style has been observed by researchers into program production and comprehension — and it can be a very successful strategy (e.g., Lange & Moher, 1989; Rosson & Carroll, 1996; see Lay, 1996 for a feminist perspective). Indeed, this bricoleur strategy should sound familiar if we think about the code in terms of genre. Students learning to write memoranda, for instance, often start out by simply copying an example's memo heading.

Project code, then, was often approached as narrative descriptions of routines and datatypes. By examining project code, developers at all three sites gained practical (although not always detailed) knowledge of relevant library code. Code was used for other things, of course — such as building programs. But these developers had developed certain ways to coordinate project code and comments to mediate their various activities related to searching, comprehending, and producing code.

Of these genres, developers mostly worked with the project code and the comments embedded within it. For the most part, they were not interested in the libraries' abstract descriptions of datatypes and routines. Rather, they wanted to see how the routines and datatypes had been used in concrete ways, for concrete purposes. Once they had seen routines and datatypes used in one or more examples in the project code — in familiar contexts, dealing with familiar problems — they generally did not need more abstract description. To use a literary analogy, developers preferred to learn vocabulary from the code's many narratives rather than from the dictionary definitions supplied in the libraries. These narratives provided context for understanding other genres, such as library code.

Although code examples helped developers by providing narratives of code-in-use, they could also inhibit code development in the long term: Copying and modifying examples could propagate mistakes and misunderstandings in the code and contribute to "code bloat" (McLellan, Roesler, Fei, Chandran, & Spinuzzi, 1998). Code examples mediated the use of libraries (and

vice versa); yet, if the examples were copied, they subverted the very purpose of libraries by introducing repeated, unneeded code.

Implications for Supporting Schlumberger's Software Developers

This informal study illustrates some of the insights that we can gain from using the genre ecology framework in studies of textual artifacts. This framework, with its focus on interpretation and its attention to compound mediation, has potential for helping technical communicators to analyze the complex interconnections among texts. In particular, the framework allowed me to compare inter-ecological relations (that is, how the same type of artifact mediates an activity differently in different ecologies) as well as intra-ecological relations (how artifacts within an ecology jointly mediate an activity).

Using the analytical framework, I was able to systematically examine the many genres that developers used to mediate their work, particularly in terms of interpretation. This informal study suggests that software development is interpretive, contingent, and especially co-mediated by a variety of textual artifacts – to a much greater extent than is typically acknowledged in studies of programmers. Furthermore, software development is exposed as a fundamentally collaborative activity that develops differently within different cultural-historical milieus and that reflects the values emergent in those milieus. Genres such as comments, for instance, indicated and shaped the values that their communities had developed over time.

Based on this analysis, the USER team was able to gain insights into how Schlumberger's developers marshall various genres to jointly mediate their programming activity. Partly in response to this study, Schlumberger's Usability Services for Engineering Research (USER) team designed three information systems to help better mediate the activity.

An automated, webbed data reference was designed to supplement the traditional printed documentation. Developers found the traditional documentation to be inadequate for developers' work because it became out of date more quickly than it can be written. In contrast, the automated reference was designed to be recompiled from the library code each night, ensuring that the online reference would always be up to date (McLellan, Roesler, Tempest, & Spinuzzi, 1998). This reference, then, was a version of the familiar reference manual genre, but one that avoided the chief drawback of that genre.

A set of complex, well-commented example programs were designed to be distributed with the code. These programs were to provide a sort of detailed narrative illustrating the use of key datatypes and routines (McLellan, Roesler, Tempest, & Spinuzzi, 1998). These examples were similar to the examples that developers already used, but were designed to exhibit consistent and preferred code sharing practices. Features that had been found to inhibit use at these sites (e.g., inconsistencies in programming style) were muted or eliminated. In this case, the genre of example code was adapted to encourage desired coding practices.

A software mining tool was developed (McLellan, Roesler, Fei, Chandran, & Spinuzzi, 1998). This tool drew on the success of genres such as automated searches and online internal documents, as well as software metrics tools and visualization tools. The software mining tool automatically scanned a product's entire baseline and created a database containing quantitative and qualitative data about the code. Developers and project managers could then query the database to get a different view of the code: Rather than looking at ground-level code or the one-line slices returned by grep queries, these users could produce a variety of reports and graphs detailing how various aspects of the code are used in the existing code. And these reports and graphs were rendered rapidly, meaning that users could form and test hypotheses about the larger system on the fly. The software mining tool combined contextualization (a feature of the example code) with text searches (a feature of grep and text editors). And it combined these with other features to provide levels of detail that were not available to developers before. Thus developers could be led to specific code without getting lost in the details. The software mining tool, then, was designed to supplement other search genres.

Future Directions

As I've argued here and elsewhere (Spinuzzi, 2001b), unlike other analytical frameworks for studying compound mediation, the genre ecology focuses on interpretation within cultural-historical activities. Consequently, I see it as uniquely suited for helping us to examine how textual artifacts jointly mediate activities – whether we are technical communicators, rhetoricians, information designers, or workplace researchers of other stripes.

As I noted earlier, analytical frameworks offer three key advantages to researchers: standardization, critical reflexivity, and scalability. But since the genre ecology framework is in its infancy, it has yet to realize these advantages. In this last section of the paper, I want to suggest some ways in which genre ecologies might be further refined in these terms.

In terms of standardization, although it is based on solid principles, this framework may benefit from more standardized heuristics. For instance, elsewhere (Spinuzzi, 2002) I suggest that genre ecology diagrams could be more useful if they were to become more formalized – if, for instance, researchers collected quantitative data about artifact use and were able to reflect those data in elements of the diagram such as distance between nodes. Standardization could lead to stronger reliability and validity.

In terms of critical reflexivity, the framework would benefit from more comparative studies with other frameworks. I have started this work (Spinuzzi, 2001b), but more can be done in comparing the framework with other frameworks grounded in cultural-historical theory such as actor-networks (Miettinen, 1999; Engeström & Escalante, 1996) and functional systems (Hutchins, 1995a; Nardi, 1996b). More critical reflexivity should lead to a more mature framework with increased analytical power.

Finally, the genre ecology framework is scalable in that it can address ranges of artifact types: It can be used to examine a relatively small number of textual artifacts, as in this study, or larger

numbers of artifacts. But the framework should become more scalable as it is refined and systematized, and particularly as that systematization leads to computer modeling of genre ecologies. (Edwin Hutchins' [1995a] work in modeling functional systems can provide guidance here.)

In this chapter, I have outlined the genre ecology framework and used an illustrative study to demonstrate its utility for studying compound mediation. I've emphasized how this framework addresses interpretation in ways that can be useful for those who study and design textual artifacts. Yet the framework can become much more useful as it is refined, filling what I believe to be an important need in workplace communication research.

References

Ackerman, M. S., & Halverson, C. (1998). Considering an organization's memory, *CSCW '98 Conference Proceedings* (pp. 39-48). New York: ACM, Inc.

Ackerman, M. S., & Halverson, C. A. (2000). Reexamining organizational memory. *Communications of the ACM, 43*(1), 59-64.

Bakhtin, M. M. (1986). *Speech genres and other late essays*. (1st ed.). Austin: University of Texas Press.

Bazerman, C. (1994). Systems of genre and the enactment of social intentions. In A. Freedman & P. Medway (Eds.), *Genre and the new rhetoric* (pp. 79-99). London; Bristol, PA: Taylor & Francis.

Beyer, H., & Holtzblatt, K. (1998). *Contextual design: Defining customer-centered systems*. United States of America: Morgan Kaufmann Publishers, Inc.

Bødker, S. (1996). Creating conditions for participation: Conflicts and resources in systems development. *Human-Computer Interaction, 11*, 215-236.

Bødker, S. (1997). Computers in mediated human activity. *Mind, Culture, and Activity, 4*(3), 149-158.

Boehm-Davis, D., Fox, J., & Philips, B. (1996). Techniques for exploring program comprehension. In W. Gray & D. Boehm-Davis (Eds.), *Empirical Studies of Programmers, Sixth Workshop* (pp. 3-21). Norwood, NJ: Ablex.

Brooks, R. (1983). Toward a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*, 543-554.

Coble, J. M., Maffitt, J. S., Orland, M. J., & Kahn, M. G. (1996). Using contextual inquiry to discover physicians' true needs. In D. Wixon & J. Ramey (Eds.), *Field methods casebook for software design* (pp. 229-248). New York: John Wiley & Sons.

Cole, M. (1996). *Cultural psychology: A once and future discipline*. Cambridge, MA: Belknap Press of Harvard University Press.

Devitt, A. J. (1991). Intertextuality in tax accounting: Generic, referential, and functional. In C. Bazerman & J. G. Paradis (Eds.), *Textual dynamics of the professions: Historical and contemporary studies of writing in professional communities* (pp. 336-357). Madison, WI: University of Wisconsin Press.

Engeström, Y. (1990). *Learning, working, and imagining: Twelve studies in activity theory*. Helsinki: Orienta-Konsultit Oy.

Engeström, Y. (1992). *Interactive expertise: Studies in distributed working intelligence*. Helsinki: University of Helsinki.

Engeström, Y., & Escalante, V. (1996). Mundane tool or object of affection? The rise and fall of the Postal Buddy. In B. A. Nardi (Ed.), *Context and consciousness: activity theory and human-computer interaction* . Cambridge, MA: MIT Press.

Freedman, A., & Smart, G. (1997). Navigating the current of economic policy: Written genres and the distribution of cognitive work at a financial institution. *Mind, Culture, and Activity, 4*(4), 238-255.

Haas, C. (1996). *Writing technology: Studies on the materiality of literacy*. Mahwah, NJ: L. Erlbaum Associates.

Henderson, P. G. (1996). Writing technologies at White Sands. In P. A. Sullivan & J. Daughterman (Eds.), *Electronic literacies in the workplace: Technologies of writing* (pp. 65-88). Urbana, Il: National Council of Teachers of English.

Hutchins, E. (1995a). *Cognition in the wild*. Cambridge, MA: MIT Press.

Hutchins, E. (1995b). How a cockpit remembers its speeds. *Cognitive Science, 19*, 265-288.

Johnson, R. R. (1998). *User-centered technology: A rhetorical theory for computers and other mundane artifacts*. New York: SUNY Press.

Johnson-Eilola, J. (2001). Datacloud: Expanding the roles and locations of information, *SIGDOC 2001 Conference Proceedings* (pp. 47-54). New York: ACM, Inc.

Lange, B., & Moher, T. (1989). *Some strategies of reuse in an object-oriented programming environment.* Paper presented at the Human Factors in Computing Systems: CHI '89 Proceedings.

Lay, M. M. (1996). The computer culture, gender, and non-academic writing: An interdisciplinary critique. In A. H. Duin & C. J. Hansen (Eds.), *Nonacademic writing: Social theory and technology* . Mahwah, NJ: Lawrence Erlbaum Associates.

Leont'ev, A. N. (1978). *Activity, consciousness, and personality*. Englewood Cliffs, NJ: Prentice-Hall.

Leontyev, A. N. (1981). *Problems of the development of mind*. Moscow: Progress.

McLellan, S., Roesler, A., Fei, Z., Chandran, S., & Spinuzzi, C. (1998). Experience using web-based shotgun measures for large-system characterization and improvement. *IEEE Transactions on Software Engineering, 24*(4), 268-277.

McLellan, S., Roesler, A., Tempest, J., & Spinuzzi, C. (1998). Usability testing application programming interfaces as input to reference documentation. *IEEE Software, 15*(3), 78-86.

Miettinen, R. (1999). The riddle of things: Activity theory and actor-network theory as approaches to studying innovations. *Mind, Culture, and Activity, 6*(3), 170-195.

Mirel, B. (1988). The politics of usability: The organizational functions of an in-house manual. In S. Doheny-Farina (Ed.), *Effective documentation: What we have learned from research* (pp. 277-298). Cambridge, MA: MIT Press.

Mirel, B., Feinberg, S., & Allmendinger, L. (1991). Designing manuals for active learning styles. *Technical Communication, 38*(1), 75-87.

Nardi, B. A. (Ed.). (1996a). *Context and consciousness: Activity theory and human-computer interaction*. Cambridge, MA: MIT Press.

Nardi, B. A. (1996b). Studying context: A comparison of activity theory, situated action models, and distributed cognition. In B. A. Nardi (Ed.), *Context and consciousness : activity theory and human-computer interaction* (pp. 69-102). Cambridge, MA: MIT Press.

Nardi, B. A., & O'Day, V. L. (1999). *Information ecologies: Using technology with heart*. Cambridge, MA: MIT Press.

Orlikowski, W. J., & Yates, J. (1994). Genre repertoire: The structuring of communicative practices in organizations. *Administrative Science Quarterly, 39*, 541-574.

Page, S. R. (1996). User-centered design in a commercial software company. In D. Wixon & J. Ramey (Eds.), *Field methods casebook for software design* (pp. 197-213). New York: John Wiley & Sons.

Paradis, J. (1991). Text and action: The operator's manual in context and in court. In C. Bazerman & J. G. Paradis (Eds.), *Textual dynamics of the professions: Historical and contemporary studies of writing in professional communities* (pp. 256-278). Madison, WI: University of Wisconsin Press.

Pennington, N., Lee, A. Y., & Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction, 10*, 171-226.

Raven, M. E., & Flanders, A. (1996). Using contextual inquiry to learn about your audiences. *Journal of Computer Documentation, 20*(1), 1-13.

Rist, R. (1989). Schema creation in programming. *Cognitive Science, 13*, 389-414.

Rist, R. S. (1991). Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human Computer Interaction, 6*, 1-46.

Rogers, Y., & Ellis, J. (1994). Distributed cognition: An alternative framework for analysing and explaining collaborative working. *Journal of Information Technology, 9*, 119-128.

Rosson, M. B., & Carroll, J. M. (1996). The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction, 3*(3), 219-253.

Russell, D. R. (1997). Rethinking genre in school and society: An activity theory analysis. *Written Communication, 14*(4), 504-554.

Schryer, C. F. (1993). Records as genre. *Written Communication, 10*(2), 200-234.

Schuler, D., & Namioka, A. (Eds.). (1993). *Participatory design: Principles and practices*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, SE-10*(5), 595-609.

Spinuzzi, C. (1999). Grappling with distributed usability: A cultural-historical examination of documentation genres over four decades. In J. Johnson-Eilola & S. Selber (Eds.), *ACM SIGDOC '99* (pp. 16-21). New York: ACM, Inc.

Spinuzzi, C. (2001a). "Light green doesn't mean hydrology!": Towards a visual-rhetorical framework for interface design. *Computers & Composition, 18*, 39-53.

Spinuzzi, C. (2001b). Software development as mediated activity: Applying three analytical frameworks for studying compound mediation, *SIGDOC 2001 Conference Proceedings* (pp. 58-67). New York: ACM, Inc.

Spinuzzi, C. (2002a, March 20-23). *Post-Process Theory and Programming: Toward a Hermeneutic Understanding of Programming Languages.* Paper presented at the Conference on College Composition and Communication, Chicago.

Spinuzzi, C. (2002b). Toward integrating our research scope: A sociocultural field methodology. *Journal of Business and Technical Communication, 16*(1), 3-32.

Spinuzzi, C., & Zachry, M. (2000). Genre ecologies: An open-system approach to understanding and constructing documentation. *Journal of Computer Documentation, 24*(3), 169-181.

Takang, A., Grubb, P., & Macredie, R. (1996). The effects of comments and identifier names on program comprehensibility: An experimental investigation. *Journal of Programming Languages, 4*, 143-167.

Tenny, T. (1988). Program readability: Procedures vs. comments. *IEEE Transactions on Software Engineering, SE-10*(5), 595-609.

Wiedenbeck, S. (1986). Process in computer program comprehension. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers: First Workshop* (pp. 48-57). Norwood, NJ: Ablex.

Wiedenbeck, S. (1991). The initial stage of program comprehension. *International Journal of Man-Machine Studies, 35*, 517-540.

Winsor, D. (2001). Learning to do knowledge work in systems of distributed cognition. *Journal of Business and Technical Communication, 15*(1), 5-28.

Wixon, D., & Ramey, J. (Eds.). (1996). *Field methods casebook for software design*. New York: John Wiley & Sons.

Zachry, M. (1999). Constructing usable documentation: A study of communicative practices and the early uses of mainframe computing in industry, *ACM SIGDOC '99* (pp. 22-25). New York: ACM, Inc.

Zachry, M. (2000). The ecology of an online education site in professional communication. In T. J. Malkinson (Ed.), *SIGDOC 2000 Conference Proceedings* (pp. 433-442). New York: ACM, Inc.

Appendix: Interview Questions

The following questions were asked by the researcher in the pre-observation interview.

Background

- What development environment are you using? (Unix, Windows, etc.)
- What are you working on, and how long have you been working on it?
- How long have you been working with C/C++?
- How long have you been working with Schlumberger's C/C++ libraries?
- Which libraries are you using the most today? In this project? In general?

Documentation needs

- How do you usually get information about Schlumberger's libraries? (e.g. asking other developers, examining the libraries, reading the documentation.)
- Under what circumstances might you use each avenue of gathering information?
- What are the advantages of getting information each way? The disadvantages?
- What types of documentation or reference materials have you used while programming in this language? About how long have you used these? What are their advantages and disadvantages?

Appendix A

Table 1: Number and Percent Words Spoken by Participant Group

Trans #	Clerk	Doctor	# Words Total	# Words Spoken			% Words Spoken		
				Clerk	Doctor	Others	Clerk	Doctor	Others
1	C6	D6	6,434	4,073	2,361	0	63	37	0
2	C6	D3	5,250	2,007	3,179	64	38	61	1
3	C2	D1	5,308	1,677	3,287	344	32	62	6
4	C2	D1	2,072	1,102	929	41	53	45	2
5	C3	D4	2,571	1,333	769	469	52	30	18
6	C5	D5	3,061	1,641	1,339	81	54	44	3
7	C5	D7	3,415	1,806	1,332	277	53	39	8
8	C7	D8	4,889	2,114	2,077	698	43	42	14
9	C8	D7	2,855	1,277	1,393	185	45	49	6
10	C8	D7	2,144	1,026	566	552	48	26	26
11	C5	D7	2,921	2,495	256	170	85	9	6
12	C10	D8	5,634	1,422	3,890	322	25	69	6
13	C10	D8	5,089	1,974	2,651	464	39	52	9
14	C12	D7	1,301	927	272	102	71	21	8
15	C11	D9	4,116	1,757	1,970	389	43	48	9
16	C13	D10	5,491	2,726	2,765	0	50	50	0
Mean			3,909	1,835	1,815	260	50	43	8

Appendix B

Table 2: Percent Words Spoken by Groups across Quartiles

Trans #	Clerk	Doctor	# Words Total	% Words (Clerks)				% Words (Doctors)				% Words (Others)			
				Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
1	C6	D6	6,434	89	79	40	45	11	21	60	55	0	0	0	0
2	C6	D3	5,250	75	41	16	21	25	59	83	76	0	0	2	3
3	C2	D1	5,308	62	34	30	0	36	64	57	91	2	3	13	9
4	C2	D1	2,072	69	33	60	41	31	58	37	54	0	9	4	5
5	C3	D4	2,571	97	54	29	27	3	27	46	43	0	19	25	29
6	C5	D5	3,061	77	63	33	42	20	34	65	56	3	4	2	2
7	C5	D7	3,415	94	76	33	8	5	23	46	82	0	1	21	10
8	C7	D8	4,889	97	43	10	23	2	52	64	53	0	6	26	25
9	C8	D7	2,855	84	47	41	6	14	48	53	80	2	5	7	13
10	C8	D7	2,144	96	68	8	20	4	32	31	38	0	0	61	42
11	C5	D7	2,921	91	97	92	62	1	2	9	24	8	1	0	14
12	C10	D8	5,634	65	15	12	9	32	76	79	88	3	8	10	4
13	C10	D8	5,089	68	41	33	13	26	58	65	61	6	1	3	27
14	C12	D7	1,301	86	77	55	68	4	13	39	28	11	11	6	4
15	C11	D9	4,116	100	48	15	8	0	48	68	76	0	5	16	17
16	C13	D10	5,491	46	50	55	41	54	50	45	59	0	0	0	0
Mean			3,909	**81**	**54**	**35**	**27**	**17**	**41**	**53**	**60**	**2**	**4**	**12**	**13**

NOTE: **Q = Quartile (1st, 2nd, 3rd, 4th)**

Notes

[1]By *informal* I mean studies that are less time consuming and relatively less rigorous than traditional academic research. Such studies are useful for descriptive and exploratory research and for guiding rapid development (Beyer & Holtzblatt, 1998; Schuler & Namioka, 1993; Wixon & Ramey, 1996).

[2] For another, shorter account of this study, see McLellan, Roesler, Tempest, and Spinuzzi, 1998.

[3] Reijo Miettinen argues: "Industrial corporations are aggregates of activity systems" (1999, p.187). He elaborates on this view in a more detailed way than this article permits.

[4]This solution is an example of *externalization* (cf. Leont'ev, 1978, p.58): once programmers have internalized the workings of grep, they can externalize it through this script, delegating the cognitive load.

[5] A *path* is a list of specific subdirectories.

[6] Cf. Bakhtin's theory of genre, especially Bakhtin, 1986.

[7]Leontyev (1981, p.406) calls this *operationalizing* actions.

[8] For a brief discussion of some of these articles in terms of code examples, see McLellan, Roesler, Tempest, and Spinuzzi, 1998.