Executable Texts: Programs as Communications Devices and Their Use in Shaping High-tech Culture

stuart mawler

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Science and Technology Studies

Dr. Janet Abbate, Chair
Dr. Barbara Allen
Dr. Barbara Reeves

19 January 2007
Alexandria, Virginia

Keywords: Programming, Culture, Code Comments, Communications, Open-Source

© 2006-2007, stuart mawler

Executable Texts: Programs as Communications Devices and Their Use in Shaping High-tech Culture

stuart mawler

(ABSTRACT)

This thesis takes a fresh look at software, treating it as a document, manuscript, corpus, or text to be consumed among communities of programmers and uncovering the social roles of these texts within two specific sub-communities and comparing them. In the paper, the social roles of the texts are placed within the context of the technical and cultural constraints and environments in which programs are written. Within that context, the comments emphasize the metaphoric status of programming languages and the social role of the comments themselves. These social roles are combined with the normative intentions for each comment, creating a dynamic relationship of form and function for both normative and identity-oriented purposes. The relationship of form and function is used as a unifying concept for a more detailed investigation of the construction of comments, including a look at a literary device that relies on the plural pronoun "we" as the subject. The comments used in this analysis are derived from within the source code of the Linux kernel and from a Corporate environment in the US.

Table of Contents

Introduction ... 1
Key Argument / Thesis ... 4
STS Literature ... 5
The Source Archives ... 11
Frames ... 14
Constructing Comments ... 21
Language ... 22
Environment ... 24
Culture ... 26
Programming Languages ... 31
Mapping Comments: Form/Function Grid ... 38
“Good” Comments ... 42
Identity-Orientation ... 50
“We” Construction ... 55
Group Elevation ... 58
Boundaries ... 64
Cyborg Community ... 69
Corporate Identity ... 73
Conclusion ... 77
Pragmatism ... 79
Future Research ... 80
Bibliography ... 81
Source Archives ... 81
Works Cited ... 81
End Notes ... 83
Vita ... 93

List of Figures & Tables

Figure 1: Graphical depiction of the relationship of *Laboratory Life* to the structure of this paper. 6
Table 1: Participating Frames in the two samples from this thesis.. 15
Figure 2: Form & Function Framework for code comments 39
Table 2: Comparison of Comment Distribution... 57

Introduction

This paper refocuses the view of software away from the utilitarian purposes of its completed form and toward its uses as both a communications medium and the central medium for discussing and defining values within computer culture. To reach its completed form, the vast majority of software is written and maintained in human-readable text (called source code), then converted by another piece of software (the compiler / interpreter) into machine-readable executable code (also called object code). In addition, software is usually not composed of a single program, but is made up of small programs and many other file types that serve specific purposes (for example, detailing the data sent from one program to another). These additional files are also written in human-readable formats and may be used in this format by the executable code. Taken together, I have dubbed any text-based file an "executable text," since these texts are intended to be used in the execution of a function or program, but are also intended to be written and read as a text. In this situation, a "text" is to be viewed with almost Biblical connotations, in the sense of a document that can be opened and read, a manuscript that is written "by hand" and pored over by other developers, or a corpus representing the shared knowledge of the group.

In 1984, Donald Knuth, an icon of computer programming, having written the TeX document presentation language and worked extensively in the Stanford AI lab, stated simply: "I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming'."[1] Knuth asked programmers to look at their job differently, saying, "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on

explaining to human beings what we want a computer to do,"[2] which is captures a small portion of the breadth of meaning implied in an executable text.

More contemporary style guides urge programmers to "code for human consumption,"[3] emphasizing that executable texts are intended to be read, as Knuth urges us to believe. Since these texts are to be read, we can conclude that the executable text is a communicative device, as would be expected of any text. Most obviously, executable texts include information in the form of the code itself (for example, LISP, FORTRAN, PASCAL, Java, C, C++, or COBOL), which will show how that program integrates with the machine. However, these texts communicate many levels of information probably unintended by Knuth's literate programming campaign.

By design, all executable texts exist in the background of the software experience, with the intention that most people be unaware their existence. The corner ATM or word processing software on a computer are examples of software that relies on many files to accomplish what looks like one task—this is software as people generally experience it, in its completed form with a human-oriented interface. In completed form, software is often a tool that enables communication, but is usually not a communications device itself. However, as a text, the source code is, itself, a communications medium, helping solidify self- and community-identities of programmers.

When programmers look at source code, the code provides information about how it functions; it acts as a repository of knowledge. In addition, most programming languages have a built-in capability to include natural language comments, without impacting the operation of the running code (often being removed by the translation step from human to machine-readable). These comments are considered "documentation," which the Linux Information Project defines as "any communicable material that is used to describe, explain or instruct regarding some

attributes of an object, system or procedure, such as its parts, assembly, installation, maintenance and use."[4]

The normative structures and processes of programming both allow and encourage individual programmers to include comments within the body of the text. While the code is considered by many to speak for itself,[5] practices encourage comments as a means to address issues of complexity and wider context that might make the code within one program difficult to understand. The Linux Information Project provides a clear explanation of comments, when they write:

> One of the most important forms of documentation for computer software is one that ordinary users rarely, if ever, see. It is the comments that are included in the source code of programs. Source code is the version of software (usually an application program or an operating system) as it is originally written (i.e., typed into a computer) by a human in plain text (i.e., human readable alphanumeric characters) in a programming language.
>
> Comments are separated from the source code by special markers and do not affect its operation. They are statements by programmers explaining their code to other programmers who may work on the same programs and to remind themselves of what they did or what remains to be done. The comments ideally include the reasons that each section of code is written a particular way and what it is intended to do
> [6]
> .

This conforms to Knuth's conception of the program as literature and the programmer as "essayist."[7] The comments illuminate specific modes of programming, complicated algorithms, history of changes to the text, or some collective knowledge of the specific business or technical problem being addressed, generally serving to make modifications, corrections, and enhancements simpler in the future—they "tell you (and any future developer) what the program is intended to do"[8] and provide this information beyond the view of most "ordinary users." However, comments serve many roles beyond the norms espoused by industry leaders like Knuth and the Linux Information Project.

Key Argument / Thesis

This paper takes a fresh look at software, treating it as a document or text to be consumed among communities of programmers and uncovering the social roles of these texts within high-tech communities. These social roles are highlighted through the form and function of comments, which include both normative and identity-oriented aspects. Comments do much more than illuminate the operation of the code. Comments, and the executable texts as whole cultural artifacts, provide a means to "discuss" issues related to programming, system design, and system maintenance, including program design style, indeterminacy with respect to code functions, and outside resources to enable solutions to future problems. Perhaps less obviously, but even more critically, executable texts serve consciously and unconsciously as means for programmers to establish and/or reinforce group and personal identities.

Within the community and the technical literature, executable texts are not simply artifacts to be interpreted into machine-readable files and then discarded. The executable texts themselves form the basis of normative structures that dictate communication of contextual knowledge (about the text and its functions) in an attempt to reduce complexity—executable texts are the foundation of knowledge in programmer discourses. The inclusion of natural language comments within the executable texts is intended to function as an aid in the creation of this foundation of knowledge, by explaining the intent of complex passages of code or inputs to the program from other processes. However, these same comments create a fertile channel for additional social interactions, often through identity-oriented forms that coexist with the normative functions. Primarily as a result of both the form and function of comments, executable texts communicate knowledge of the world outside the program; they transfer expertise, educating other programmers about best practices; and they offer a forum to discuss

these issues. In short, they help define personal and group identities within high-tech communities.

While this paper focuses on the relationship between the natural language comments and the communities of programmers who write them, it also explores the relationship of the comments to the programming languages within which they appear. There is inherently a sense of dualism between the natural language comments and the symbolic logic of the programming language. Each is a language that is expressive for some purposes, but not for others. However, the natural language comments are included, by convention, as explanatory of the text in the programming language.

STS Literature

This work draws on many sources in order to form a broad picture of the context of natural language comments. Bruno Latour and Steve Woolgar provide the concept of the inscription device, which is analogous to the executable text. Paul Edwards, Barry Barnes, Michael Mulkay, and Thomas Gieryn, together provide a rich framework for the social and power role of discourse, while Sherry Turkle and Donna Haraway bring up concerns of identity and cyborg structures, in particular. Finally, Wiebe Bijker introduces the concept of the technological frame.

In *Laboratory Life*, Bruno Latour and Steve Woolgar assert that “the inscription device,” including reports and virtually all written records in a laboratory, is the critical product of the scientific endeavor.[9] While they acknowledge that the scientists under observation are isolating peptides and making other discoveries, they note that the majority of the lab output is actually inscription devices (primarily articles and papers). Latour and Woolgar identified two competing functions for the inscription device: end product of the scientific endeavor and

communications device inside and outside science. The novelty of this approach is in isolating the inscription device as the product of science, rather than solely as a vehicle for communication. For Latour and Woolgar, the production of textual artifacts becomes paramount.

To a much greater degree than Latour and Woolgar's scientists, the work of programmers within an IT department focuses "either directly or indirectly on documents,"[10] where the documents in question are executable texts. Unlike Latour and Woolgar, the executable text (inscription device) is already accepted as the actual product, not just a by-product. The new ground covered in this paper is in considering the executable text to be a communications device, rather than solely an end-product, thus turning the context of *Laboratory Life* on its ear (see Figure 1).

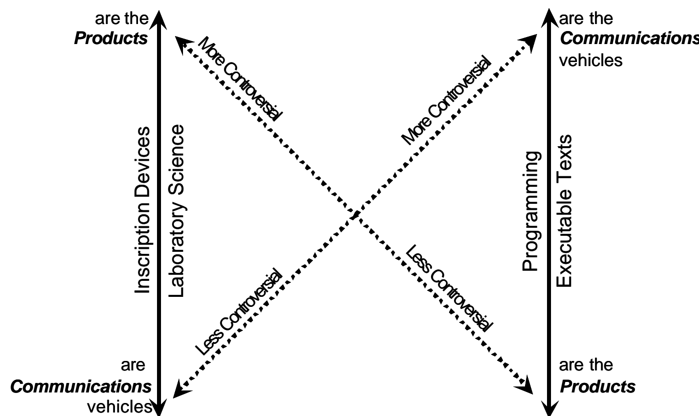


Figure 1: Graphical depiction of the relationship of *Laboratory Life* to the structure of this paper.

Latour and Woolgar also look at the public consumption of inscription devices. Many are intended for wider public audience, but still contain language that targeted them toward an audience literate in the concepts contained in the articles. Executable texts share the requirement

of technical literacy, but their target audience is limited to other, similarly literate programmers. In many corporate settings, the management is unable to read the executable texts composed by its own staff. The limited audience is further restricted in a corporate / proprietary setting by security requirements, dictating who may actually open a particular file, increasing the group-specific role of executable texts in a manner not typical of most scientific texts. In the open-source community, anyone can, in theory, read the source code (hence "open source"), but the practice is limited by literacy of both the language and the tools needed to access the source archives.

In another parallel with *Laboratory Life*, where "only a few of [the reports] appear in published form,"[11] not all executable texts need enter production. Computer code may be written for teaching purposes, as tests of ideas, or as projects that are abandoned for any number of reasons external to the act of programming. Rather than constraining the definition of executable text, the lack of production use supports it, as these texts have no direct machine-related purpose at all. In short, by looking at comments in the executable texts, I invert Latour and Woolgar's concept of the inscription device, showing how technical and social practices are embodied within the code, making the executable text not only the product of programming, but a key communications vehicle and support for the community.

As a communications device, the executable text is the embodiment of programmer discourse and, as the primary product of the programmer activity, forms the central Foucauldian support for the programmer's "structure of production and exchange of useful things," as Paul Edwards notes in *The Closed World*.[12] By creating comments, following standard practices, programmers embed knowledge into the executable texts. Where comments extend beyond their normative purposes, in either form or function, they more clearly take on the support function.

In the participant's view of programming culture from within, the identity of a programmer is usually known by his/her abilities; a programmer who quickly writes clean, readable, efficient code will be known for his/her work. However, simply writing good code is not enough; the efficiency needs to be advertised; bad decisions outside of a programmer's control need to be highlighted; the inelegancies required of a particular situation must be noted. Comments serve all of these functions and more, helping to drive the ascendancy of particular programmers, groups, or even theories and methodologies of programming, in a process that is not static and not driven from the top. Even in corporate programming situations, the developers spend their time in the executable texts, while management does not, so the programming discourse is built from the bottom up.

While Edwards tends toward a single overarching discourse (seen as "closed world" or perhaps cold war), programming, as a whole, is comprised of many competing discourses (for example, appropriate programming techniques, appropriate documentation techniques), which all end up finding a voice within the executable text. The executable text "is the object at once studied and invented by the discourse[s] that surrounds it."[13] In no discipline is this more true than programming, where the practitioners create their support in the form of the executable text and then leverage it for the creation of additional texts that expand, continue, and enrich the overall discourse.

The normative practices of programming, which advocate extensive use of comments, become key parts of the discourse. The inclusion of comments furthers the adherence to the norms, creating an ideological basis for this form of "knowledge" in the sense used by Barry Barnes, where knowledge is "accepted belief and publicly available, shared representations."[14]

The executable text acts as that shared representation, which is available to the community of programmers.

Importantly, Barnes also allows us to view comments as an attempt at control. Programmers, as much as scientists, operate "in terms of an interest in prediction and control shaped and particularised [sic] by the specifics of their situations,"[15] where the situation is the wider system for which they are programming, and an expected outcome needs to be predicted and controlled. Even though the code is written to create specific outcomes, the system might have unintended results (for example, a bug). Even expert programmers can and do disagree over how best to write a particular function in a program or system (which will be seen in the comments). Essentially, programming becomes non-deterministic and unpredictable with increased complexity, allowing comments and their structure to provide a feeling of control over a highly volatile knowledge base. Adherence to the practice of commenting supports the ideological approach to programming that advocates the use of comments.

Michael Mulkay writes about "vocabularies of justification,"[16] where specific terms are used for support of an ideology. In the case of programming, the comments themselves become part of the vocabulary of justification. The success of the device is attested by its existence in the code. Similarly, the existence of comments in the code supports the belief in their efficacy. This circular logic points to what may be an underlying belief held even by programmers in the supremacy of natural language over symbolic logic as a descriptive tool for any situation, regardless of complexity. Natural language comments are inherently considered to be more clear than code because they are "regular" or "normal" language. Further, the structure of comments often provides authority to the speaker, in such a way to extend beyond "vocabularies" to grammars of justification. Vocabularies of justification imply a collection of specific words or

phrases that are used to support a viewpoint in a given social context. Expanding the concept to "grammars" opens the possibility that sentence structure and form can have ideological implication. For example, in the Linux sample, the choice of plural pronouns in the sentence subjects serves identity-oriented goals in a subtle, yet pervasive way that might be transparent to readers, since it does not impinge on the overall explanatory function of the comment.

While the intent of the comment capability may have originally been an explanatory aid, comments have emerged into a world where they are supports for ideologies used in the "pursuit of authority and material resources" as Thomas Gieryn describes it: to expand authority into other domains, highlighting contrasts; to monopolize authority and resources, excluding rivals as outsiders; and to protect autonomy, blaming outsiders.[17] Programmers are defending their turf through comments. Their boundary struggles are many—with management, with other groups within a corporation (for example, Marketing and Finance), with competing methodologies, with other elements of the industry, and with user groups. Here again, it is the form of comments that points toward these alternative interpretations.

Comment authorship also occurs in a world and a discourse, where the definition of self is fluid. As Sherry Turkle points out, computers "stand in a novel and evocative relationship between the living and the inanimate. They make it increasingly tempting to project our feelings onto objects and to treat things as though they were people."[18] Donna Haraway directly acknowledges this struggle, taking it to the next level, when she states that she "has repeatedly tried to make problematic just what does count as self, within the discoveries of biology and medicine, much less in the postmodern world at large."[19] The form of the comments found in this study shows that this struggle extends to computer programming, where the postmodern authors of executable texts do more than project feelings onto the systems they create—they

associate directly with the machine, creating bodies and identities that are constructed not merely through biomedical discourse, as Haraway suggests,[20] but through programming discourse.

While both the normative and identity-oriented functions of comments in executable texts are similar, regardless of the text, it is important to note that not all comments are created in exactly the same way. Wiebe Bijker offers the concept of the technological frame, as a means to describe the set of assumptions and approaches that an individual may use with a technological problem. In programming, technological frames critically include the platform on which a program is built (for example, mainframe versus personal computer), the language in which a program is written (for example, C or COBOL), the purpose of the software being built (for example, an operating system or a corporate customer database), and the business context in which the application is being written (for example, open-source versus proprietary).

The Source Archives

To complete this study, I have relied on two collections of executable texts from vastly different technological frames; the Linux kernel and a Corporate sample from a proprietary environment.[21] Linux is an open-source operating system, meaning that the source code is available freely on the internet and development occurs collaboratively across a wide, and voluntary, community. Despite being a voluntary development team, new software versions are as tightly controlled as in proprietary software development.[22] The kernel has been selected for greater attention since it is a key portion of the operating system, handling interaction between the system and the hardware itself, ensuring that all processes receive the hardware resources they require in the order established by the rules of the operating system. The criticality of the function within the operating system helps ensure that there will be healthy debate around the specific programming tasks. Within the Linux kernel, I have chosen to further pinpoint my focus

on 12 specific executable texts of varying lengths, using comments excerpted from the texts to illuminate and provide evidence for the various themes of this paper.

The Corporate sample comes from an internal IT department supporting the consumer customer database of a large US-based corporation. There are seven texts in the set and all seven are COBOL modules, intended to be executed on the mainframe in IBM's CICS/MVS, where the texts were originally composed and all subsequent maintenance occurs. The centrality of the customer database to business goals, coupled with the age of the system guarantees healthy debate like that expected in the Linux kernel.

Looking at each sample, comments are considered to have both form and function, where each is a continuum from normative to identity-oriented. The normative end of the scale contains "good" comments, which are intended to enable understanding of the program at hand, as recommended by the style guides. The latter end of the scale more directly serves the construction and reinforcement of programming community and discourse. Between each of those extremes, comments may illuminate issues or information not necessarily central to the executable text at hand, providing a means to share information between developers across time and space.

Within this framework, the examples highlight many social factors. The two samples represent competing technological frames, while containing multiple active frames within each sample. With respect to the comments, both samples highlight the metaphoric status of programming languages. However, the most important aspect of the comment analysis is how it highlights the social nature of comments and their role in reinforcing community.

While in both samples the comments can be represented across the same normative to identity-oriented continuum of form and function, the technological frames strongly influence

the particular forms implemented in each sample. Within the Linux kernel, the comments show several identity-oriented forms, where the language is either explicitly or implicitly structured around the inclusive pronoun "we," with various implications. In contrast, literary structure is largely absent from the Corporate sample, where the form of the comments tends toward more "aggressive" literary forms, dominated by imperative verbs, with a different type of culture being reinforced, reflected, and created as a result.

Frames

As a whole, each sample (the Linux kernel and the Corporate sample) can be seen to represent the second of Bijker's technological frames: a single dominant interest group or technological frame, where training in the frame is widespread and effective, but group members on the periphery can think outside the frame.[23] In the Linux example, the dominant frame is one of open source programming of operating systems. The corporate example represents a dominant frame of mainframe corporate programming in COBOL. My use of multiple descriptive words hints at the fact that, similar to discourses, there are actually multiple competing frames operating at any given time within each overall framework (see Table 1). But there is one basic concern that exists before any discussion of the subtlety of languages, platforms, or environments: programmers are both consumers and creators software.[24]

Programmers are in a situation that is unique in controlling the outcome of the technologies and being a user of them at the same time. The dual role is all the more important since programmers do not see themselves as users. To be a member of the class of users is to be essentially non-technical. Even the unacknowledged confluence of frames creates tension, in that the programmers view these frames or roles as inherently in conflict.

Importantly, while the junction of user and creator makes programming somewhat different from other disciplines, it also separates the two samples. For the Corporate sample, the conflict is reduced somewhat by the filter of corporate life: this is a job to which one commutes and for which one gets paid. Further, the product of the programming is used behind the scenes at a corporation, where there is no ability to directly interact with it.[25] In the open source example, the programmer is, in many ways, the target user, making separation of the user and the creator much more problematic. In neither case, however, does the programmer associate with

the user directly. The programmer is more likely to be influenced by the intersection of other frames as factors in his/her identity and community, retaining a general sense of superiority to their users.

Frame:	***Linux kernel***:	***Corporate sample***:
Language	C	COBOL
Software Purpose	Operating System	Customer Database
Platform	PC / Small to Medium Scale	Mainframe / Extremely Large Scale
Development Environment	Heterogeneous (developer decides his/her environment)	Homogenous (corporation decides on environment)
Team	Single Team for all tasks	Production Support & New Projects handled separately
Programmer Relations	Extended Network / Computer Mediated	Closed Network / Face-to-face
Ideology / Economics	Open Source / Free	Proprietary / For Profit
Power / Control	Bottom-up (self-organizing community)	Top-down (corporate hierarchy)

Table 1: Participating Frames in the two samples from this thesis.

While the list in Table 1 is not exhaustive, the frames highlighted at work within the samples cover a wide range of issues, from the choice of language to the mechanisms of power and control. Within the Linux kernel alone, an individual developer may have varying levels of participation and relationships to each of the frames, though each frame is dominant in its area. For the Linux kernel, all programming is completed in C, but developers may have varying levels of expertise in this language, as well as varying levels of exposure to other languages. More importantly, the choice of C for all of Linux represents a radically different perspective

from that represented by the choice of COBOL in the Corporate sample, where C is considered a more technical language than COBOL, which is viewed as antiquated and lacking in power.[26]

While the C programming for Linux will be used in an operating system, C is not exclusively used for operating systems, also serving application development needs in private industry and government. Further, operating system development is more technically challenging than many other forms of development, since the completed product will be the software that manages other pieces of software. It might even be possible to investigate a more finely-grained frame of kernel programming, as a sub-set of operating systems, since this area has special considerations for hardware management. By contrast, the Corporate sample is an application that relies on an underlying operating system (not Linux). Interestingly, in terms of importance to the user community, both samples may occupy the same space, since customers, who are managed by the Corporate sample, are considered a company's most important asset, but this does not seem to translate to actual attitude of the programmers in any directly detectable sense.

For Linux, the platform where the software is both developed and run is usually a personal computer (though the operating system is being used on larger and larger machines).[27] In these arenas, the scale of processing has limits. The Corporate sample is designed to handle millions of customers in an around-the-clock, year-round environment with gigabytes of data. Interestingly, while the comments reflect a sense of ambiguity around the outcomes in both cases, the comments in the Corporate sample seem to reflect a higher degree of variability in the outcomes and understanding of the process, which may be due to the greater longevity of the Corporate sample than the Linux kernel.[28]

While the development environment for Linux is the personal computer (PC), each developer has the final choice about what software s/he will use to assist them in editing the executable texts. There are many types of software development environments and each is tailored to a particular style and set of needs, with many being freely available and attractive to Linux developers for that very reason. Some developers prefer an extensive environment, which offers more real-time (as you type) assistance and greater debugging capabilities, while others prefer something more like a simple text editor, which might present the source code in monochrome or possibly with key words highlighted.[29] With this many choices and no centralized control, the decision will be guided by both personal preference and community norms, remaining fluid across the participants, potentially reflecting the influence of still other frames that are not directly evident in the executable texts themselves.

By contrast, the Corporate sample is created and maintained on company assets that are purchased and maintained by the corporation. There are compelling reasons for the company to control the software in use for the development process, with the basic reason being costs, which can include software, hardware, support, and training. The corporation might buy many copies of a single piece of software for volume discounts or purchase a much larger suite that contains software development tools for a package deal. Whatever the cost containment mechanism for purchasing, the resulting standardization enables the corporation to standardize on the required hardware and reduce the hardware and software support costs. Finally, the standardization makes training new staff much simpler, as the entire staff has the same requirements. Developers can more easily assist each other, since they use the same software. Some tasks, processes, or procedures can be enforced in an automated way through the software itself, reducing reliance on training and human intervention. Despite these clear differences, the

comments in the Linux kernel and the Corporate sample do not represent identifiable polarities of heterogeneous style versus automated regularity, respectively. In fact, while the comment form is more highly standardized in the comments found in the Corporate sample, the standardization is driven entirely through non-automated community norms.

The form of comments may be related to the structure of the teams working on each archive. In the case of the Linux kernel, there is a single development team. The people on the team focus on areas of expertise, with some members handling only a particular function (and associated set of executable texts). If an entirely new release is undertaken, it is done as a team, with the appropriate members working on their areas of expertise. If a bug or error is detected in a particular area, the team members with the appropriate expertise (which may be the entire team) address that bug. Alternatively, the bug may be cause for a new developer to join the team.

The Corporate sample represents a frame that downplays "ownership" of particular functions and executable texts, having separate development teams for new projects versus bugs and errors. The management structure is aligned to track development not by areas of expertise and knowledge, but by the type of request. This alignment is reflected in the Corporate sample's comments, which make reference either to projects or to bugs and errors, which are termed "production problems." There are absolutely no analogous references in the Linux kernel comments.

Even though there is essentially one large team contributing to the Linux kernel, that team is highly extended in structure. The developers can be anywhere that has an internet connection and may never meet each other physically; their regular interactions are almost completely mediated by computer-based communications technology. Further, admission to the

team is by community agreement and demonstrated ability; anyone may join, regardless of certifications or other formal criteria, with their continued participation hinging on peer reviews of their executable texts. This team structure may be radically different from the manner in which they conduct their day jobs and is almost certainly radically different from that used to develop other proprietary operating systems. However, not all developers may participate to the same degree, as some are employee of companies that sell support for versions of Linux[30] and many may actually know each other and interact outside computer-mediated environments.

The Corporate sample was created and maintained by programmers in a closed group. Admission to the frame requires authorization from management through a formal application process, usually including a resume and interview, emphasizing formal criteria like education and certifications. Also notable is that the frame implies, for the most part, being physically co-located with the other members of the team. Though the team clearly relies on computer-based communications (as the very existence of executable texts testifies), decisions regarding development options and issues are addressed face-to-face. This frame is more traditional than the geographically diverse and computer mediated frame dominant in the Linux kernel development, but again, not all members participate to the same degree, with some placing greater or lesser emphasis on some aspects like formal certifications or face-to-face interaction.

The ideology and economics of Linux kernel development form another non-traditional, but extremely important frame, with potentially the greatest impact on comment form. This frame informs many of the many community practices through the emphasis on the idea that information should be free. Even here, the participants may have varying degrees of buy-in to the ideology. Some team members may have joined through a purely ideological intent; some members may be driven by an anti-Microsoft ethic; others may be driven to create interesting

code for problems different than those found at their day jobs; while still others might simply have acquired a job with a software company like IBM, where contributing to Linux is a portion of their job description. However, the last category is likely the smallest, with most team members actively seeking out the open source environment for some ideological reason. Since the team is largely volunteer and the product is largely free (unless you pay a company like Red Hat, Inc for their add-ons and support), a more collegial atmosphere seems a likely result; participants are engaged by their own choice and wish to attract and retain highly qualified members, so an adversarial tone in the executable texts would be counter productive, and this is exactly what the comments reflect.

While the open source philosophy consciously frames the development within the Linux kernel, the Corporate sample is just as strongly influenced by the ideology and economics of proprietary software created for profit. In this sample, the language is COBOL, the hardware is a mainframe, and the operating system is CICS/MVS, which helps to amplify the for-profit nature of the programming tasks—the environment is not considered new and exciting, they are simply tools to complete a job. Those contributing to the solution may be doing interesting work, but this is much more likely to be a source of income rather than a life's calling. But even in this setting, some developers are intrigued by solving complex software problems; some prefer large mainframe environments; others thrive on the urgency and hectic pace that come with development in a for-profit, market-driven environment. However, when the economy is good, it is possible to change jobs, so creating a collegial work environment is not as critical to the individual developers.

The ideology and team structure together imply the power and control dynamics of the communities. The Linux kernel is built by a volunteer team, where membership is based largely

on ability and performance, resulting in a power structure that is more flat (or peer-to-peer), while the Corporate sample has a team built by management with profit and regulatory concerns, resulting in an overall top-down structure. The collegial tone of comments in the Linux kernel emphasizes the lack of a clear hierarchy, where a central authority can give direct orders to other members. The Linux kernel development, however, maintains a leadership and a core group of developers with more authority than others, creating a form of hierarchy, even if that structure is based originally on performance. Even at the union of ideology and team structure, the power structure frame may have varying levels of buy-in, since some members may prefer greater centralized control than others, or some tasks may require greater standardization.[31]

Within the Corporate sample, the form of the comments is much more imperative, like an order being issued, rather than a discussion or a lecture, complementing the top-down tone of corporate development. Despite the control associated with a corporate environment, there are levels to which employees conform to the power dynamic of the corporate hierarchy. The lack of conformity is evident in the content of the comments. It is in the comments that the individuals and the group see expression, since the individual developers control their content and discussion within the executable texts.

Constructing Comments

While these, and arguably other, factors collaborate to construct the two overall frames represented by the two archives, there are specific considerations that go into the construction of comments. At the most basic level, there is the normative intent and purpose of comments, but there are also very specific considerations driven by a combination of the language used (C versus COBOL, mapping to the Language frame in Table 1), the environment (PC-based versus mainframe-based, mapping to both the Platform and Development Environment frames in Table

1), and the culture (open-source versus proprietary, focusing on the Ideology / Economics frame in Table 1) in which it is used. These format considerations apply to all categories of comment.

Language

Though almost every language retains the ability to include natural language comments along with the symbolic logic of the program code, the format of the comments is different in each. In C, comments all begin with a "forward slash" ("/"), followed by an asterisk ("*"), and are concluded by an asterisk ("*"), followed by another forward slash ("/"). All information between those two sets of marks will not be included in the operation of the program. While the compiler (the program that converts the symbolic logic in the text from source code to object code) only requires the starting and ending marks, individual developers may choose to use an asterisk on the left side of every line (as is the case in most of the multi-line comments used as examples in this paper). This is a purely visual device to make human identification of the comment quicker and easier.

Comments within COBOL are signified by an asterisk ("*") at the start of the line (which will be numbered sequentially by the environment, unlike C, which has no embedded line numbers). All information on that line to the right of the asterisk will not be included in the operation of the program. While only one asterisk is needed to remove a line from the production run of the program, more than one may also be used.

By social convention in most languages, ahead of all the symbolic code is a section that describes the basic content of the module, sometimes providing short summaries of major changes, encapsulating the program's history within the executable text. The practice of beginning each module with a short description is widely adopted across many programming disciplines, including the open source community, where one coding standards document for

work written in C says, "Every program should start with a comment saying briefly what it is for."[32] In most programming languages, these opening short descriptions are formatted as comments and neither C nor COBOL are exceptions, so the top of most programs will include commented lines, but comments may appear in this format anywhere within the program. I will call comments made within the body of the program "inline comments."[33]

In C programming, comments are often included to the right of symbolic code that will be processed by the computer. Other languages, like Fortran and COBOL, generally devote an entire line to a comment.

In addition to creating space for natural language remarks, in COBOL the asterisk is used for a wide variety of functions. Paragraphs and sections in COBOL are numbered for machine access, but are often visually highlighted by rows of asterisks above and below the section name, as in this example:

```
00084 *****************************************************************
00085 *             L I N K A G E     S E C T I O N                    *
00086 *             -------------     -------------                    *
00087 *****************************************************************
```
[34].

The asterisk can also be used to prevent execution of a line of otherwise machine-ready code, as in this example:

```
02310 *     IF  VOIP-SERVICE-PRESENT
02311 *     AND (FINANCIAL-CANCEL-RSN OR LEC-FINANCIAL-CANCEL-RSN)
02312 *         SET TAKEDOWN-VOIP-SVC        TO TRUE
02313 *         SET LG1-REQ-DEACT-VOIP       TO TRUE
02314 *     END-IF.
```
[35].

In the above example, the logic represented will not perform or compile because the lines begin (after the line numbers) with an asterisk.

Environment

Creating and maintaining the C source code for Linux is done in a wide variety of ways, according to the preferences of the individual developers. The commonality is that each programmer has the choice to apply "graphical" editors to the code. These graphical editors can be configured to highlight comments (among other items) in different colors, helping to separate them from the symbolic code. This graphical highlight makes the use of comments on the same line with symbolic code easier to manage than in other environments, where all the characters are presented in the same color and font (for example, on mainframe computers). Despite this, many programmers continue to separate the comments using asterisks for visual emphasis, perhaps as a hold-over from earlier, monochromatic paradigms. This emphasis on visual clarity is even found in recommendations available on the web, as in this example for C++:

```
//============================================================//
//Development By : Jigar Mehta
//Date : [ & now() & ]
//============================================================//
```

.[36]

In the author's recommended format, the top and bottom lines are visual only; they are lines of equals signs ("="), emphasizing that the comments are for human visual consumption—the quicker and easier to detect the better. The middle two lines carry the developer name on one line with date/time on another line. In C, this would probably be a line of asterisks, rather than equals signs, since the language requires the use of asterisks, while C++ does not.

In contrast, creating and maintaining COBOL programs in the mainframe environment offers programmers an interface paradigm (in the Kuhnian sense) that has been abandoned by almost all other disciplines. In this environment, there are no graphical user interface tools beyond monochromatic text on a monochrome background. Lacking colors or other devices,

comments, section headings, and any other notable items must be highlighted with text characters set off from the code by an asterisk at the start of the line, as in this example:

```
00577 ********************************************************
00578 *     COPYBOOK:      filename2                         *
00579 *     DESCRIPTION:   TARIFF MODULE INTERFACE FOR       *
00580 *                    MODULE ' filename1 '              *
00581 ********************************************************
00582  COPY filename2.
```
[37].

In the above example, there were many choices for presentation of the information. First, there was a decision made to present additional data, at all. Line 00582 tells the machine and the programmer that a particular copybook is being used; however, since it does not tell the purpose, the programmer decided to include that information. Second, the information could have been presented as a single line immediately above or below line 00582, or even on the same line to the right of the period. In this case, the developer felt the additional information important enough to highlight visually.

However, the mainframe environment presents an additional technology-specific constraint: screen "real estate." The example above is visually arresting and easy to see when browsing through a long program, but it also occupies five lines on a screen that only displays approximately 30 lines at a time, depending on user configuration. This tension between information and screen real estate results in some extremely terse comment styles, as in this example:

```
02871 ** MAC 07/17/95 METEOR 2QTR - BEGIN
02872       MOVE A-ACN-PRD-DATE OF RECORD-BUFCX09
02873                              TO COM-ACN-PRD-DATE.
```
[38].

The above example shows several pieces of information about the lines of code that follow it. From line 2871, we can see the identity of the programmer (MAC), the date

(07/17/95), the project that caused the edit (METEOR), the release that included the change (2QTR), and the fact that this is the start of the updated area (and, hence, to expect a very similar "end" comment in a few lines, showing the extent of MAC's impact on this paragraph). In contrast, look back at the C++ format for an inline comment referenced above, where the author recommends a using four lines to accomplish roughly the same goal, with less data.

The C++ recommendation occupies far more visual real estate than that used in the COBOL example, while providing less contextual information, since it does not include references to the project or the specific release. The author's choice of technology likely influences his formatting decision. The online article addresses C++, which is usually written in a visual editor, where comments can automatically be detected and highlighted with a different color. In addition, more lines can appear on the screen at any given time by changing font sizes or screen resolution. None of these changes to the developer work station have any impact on the number of visible lines on the mainframe text editor. When many edits overlap in a single area, four lines per edit would occupy valuable screen space needed to read the code itself.

Culture

The cultural environment provides potentially one of the largest impacts on the communication style of comments. The online author who recommends the C++ inline comment format above said that the inline comments are more critical "when we work on a big project and the work is done by more than one member of the team."[39] In the case of both the Linux kernel and the Corporate sample, the environment stresses the definitions of both "big project" and "team."

In the Linux context, the entire operating system can be considered the "project" and the entire collection of developers across time and space can be considered the "team," and the

development team, by the nature of open source, can contain members from anywhere on the planet. Further, the development team includes the originator of the project, Linus Torvalds, whose name is the basis of Linux.

The Corporate sample consists of programs that are all ten or more years old, with one having been written in 1982. The average number of lines of code per program is near 2000, with the oldest program having 15769 lines. Further, these seven comprise a tiny fraction of the number of programs, copybooks, JCL members, PROCs, DOCs, and other executable texts necessary for the proper function of the environment in which they were written. In this context, the entire enterprise can be considered a "project" and the entire collection of developers across time can be considered the "team," since the culture of the environment has been constructed across that entire unit. Interestingly, the staff turn-over within the group has, until recently remained relatively low, with tenures measured in decades, rather than years.[40]

In the Linux kernel context, the most important influence on the form and tone of comments is the collaborative and voluntary nature of open-source development. Unlike a proprietary environment, Linux developers take on tasks because they feel strongly about the goals of the project or the intellectual stimulation they receive from their tasks; it is of ideological or artistic importance to them, rather than just being a job. As a result, there is a generally congenial tone within the comments, even where there are debates about the appropriate method to code a particular passage.[41] Out of the set of programs, I found no instances of overtly antagonistic language, while such language was common in the Corporate sample.

Another culturally specific practice is the use of comments to mark the start and/or end of a change to the symbolic code with what might be considered begin/end tags. Following this

practice, a programmer would create a comment immediately above the line(s) of code to be changed and possibly one directly below the last line of code to be changed. In the Corporate sample I have found this practice quite common, but it does not exist at all within the Linux kernel.

Further, there is the culturally determined practice of handling code that is no longer needed. In some programming cultures, obsolete code is retained by making it into a comment. The old code is thus visible to later programmers, but the compiler does not convert it into object code for the computer to use. Commented code can, in theory, simply be "uncommented" to be resurrected and adds a sense of history to the text of the program. However, the concept has been successfully parodied as a counter norm in the online guide "How to write unmaintainable code," where the author writes that programmers should "be sure to comment out unused code instead of deleting it and relying on version control to bring it back if necessary. In no way document whether the new code was intended to supplement or completely replace the old code, or whether the old code worked at all, what was wrong with it, why it was replaced etc."[42]

What the commented code also reflects is uncertainty about the operation of the program and uncertainty about direction—the business direction might require that code in six months, so don't waste it. However, if you extensively document what business reason drove the code elimination and what the code was originally doing and how its removal made sense, then people can make a rational judgment about resurrection or debugging at some future date. Otherwise, the commented code is just chaff that makes a subsequent developer's job more like that of an anthropologist or archaeologist.

In the Linux kernel, there are absolutely no examples of code commented out, which does not indicate a seamless development path free from re-writes. Rather, the lack of code that has

been commented out points to a stylistic and ideological decision. The old code no longer represents knowledge for the Linux kernel contributors and, to them, its presence would simply add confusion. To view the changes over time, a person would need to compare previous versions of the same file, or use the tools of version control, as implied in the quip about unmaintainable code. While removing the old code does tend to lend clarity, it also might create a false sense of inevitability, as though the edits needed to bring the code to its current state were less extensive than they might actually have been.

In the Corporate sample, it is the size and scope of work, together with both the hierarchical nature of the environment and the environment (linguistic and technical), that most strongly influences the form and content of the comments. In the inline example above, the developer limited his/her use of space, while tersely expressing the maximum amount of content, for example, in the COBOL comment, we see the initials of the developer (MAC), instead of the full name as the online author recommends.[43] In this context, no further self-identification is necessary, since the staff is close-knit and relatively static over time. While not all inline comments conform exactly to this example, it is strongly representative of the genre.

Also of note is the inclusion of the "project name" (METEOR). Outside of the environment, the name is meaningless, but it conveys valuable contextual data. From the project name, it is possible to reconstruct the team responsible, recall overall issues, and make a quick judgment about the soundness of the IT and business decisions associated with the project. The project name is a strong cultural marker with layers of meaning.

Another cultural influence on the comment style is program length—in COBOL, programs are often encouraged to be longer, though this is purely by social convention. A single program is divided into many sub-sections, each of which may consist of many lines. In C++,

and other object-oriented languages, "elegant" programming encourages (purely by convention) the creation of many smaller programs (classes). In a longer program with more changes, it might be more important to know both the beginning and end of the change, rather than just the beginning, since there are more lines that might be changed with any one edit. Hence, in the Corporate sample, we see inclusion not just of inline comments marking the start of an edit, as the online author suggests, but a corresponding inline comment marking the end of the same edit, as in this example from the same developer, date, and project, but different program location:

```
** MAC 07/17/95 METEOR 2QTR - BEGIN
[... 14 lines of code removed ...]
** MAC 07/17/95 METEOR 2QTR - END
```
[44].

To many developers, the difference in length between COBOL and C++ would seem technologically determined by the language choice. However, nothing prevents a C++ class from being as long as any COBOL program and only convention prevents this being a regular occurrence.[45] The point of the convention is to limit the number of functions performed within a single class, thus limiting the number of changes that need to occur within a single program in any single release. The accepted wisdom is that fewer lines means fewer functions, which means fewer changes, which means fewer developers in a given program. Where there is a significant number of any of these, it might become more important to know where an edit stopped in addition to the start location.

To illustrate the social nature of the program length convention, we can look at a comment in a particular open-source Java program. In Java, an object-oriented language in the C++ tradition, a class should, ideally, represent a single business function, with a whole application (for example, a game) being comprised of several small classes. However, developers can consciously decide that the convention toward smaller and more numerous

classes is inappropriate for their particular context. The author of one on-line open-source game writes at the top of his program:

```
// Note:  I have included the entire program in one file (contrary to
// recommended Java programming guidelines) as I believe this is more
// convenient for distribution. [… ed.: text removed …]
```
[46].

Programming Languages

Having explained the structure of the comments, I will return to the programming languages themselves. As other authors have implied regarding natural languages, a "programming language" may be literal or it may be metaphoric, depending on one's point of view.[47] More importantly, a programming language may be a metaphor for the social organization of a community of programmers.

A programming language has syntax and grammar. A programming language is expressive of many complex and nuanced concepts, allowing different authors to express the same concept in different ways. All of this evidence points to a literal rather than metaphoric interpretation of the term "language." More importantly, a shared language is one of the foundations of an established community and the programming language is the most basic shared language of a given programming community.

However, a more detailed review of the structure of programming language begins to look metaphoric. When looking at specific language, programmers adopt the terms of natural language, speaking of "sentences," "paragraphs," and "verbs." Even more notably, some languages use terms like "copybook," referring to entities outside of the program that are essential to the system.[48] There are also key terms within the language that derive from natural language colloquialisms, like "jiffy," which is used in C to denote the smallest increment of time

on the system clock (about 1 millisecond).[49] Outside of programming, a jiffy is a widely variable time frame that is completely context dependent and lacks the kind of precision required by programming, while within the computer, a jiffy is always the smallest division of time. Where the two worlds overlap is in the indeterminate length, since, in C, a jiffy is not always the same length, being completely dependent on the CPU, but the indeterminacy within the computer is within a millisecond, while the real-world indeterminacy may be minutes, hours, or even days. There are also "jitters," which are "abrupt and unwanted variations of one or more signal characteristics, such as the interval between successive pulses, the amplitude of successive cycles, or the frequency or phase of successive cycles," according to US Federal standards.[50] While possessing a long history in communications, a jitter is also widely known as an uncontrolled variation that people get when having too much coffee, which seem singularly appropriate to the context of programming. Words like jiffy and jitter allow programmers to reach back into natural language to make the programming language more accessible and more socially relevant.

Some theorists are agreed that "'a good programming language is a conceptual universe for thinking about programming',"[51] making the case that a programming language is capable of expressing all the needs of programmers. However, metaphorical borrowings like jiffy and jitter seem to say that, contrary to the conceptual universe idea, "a program cannot speak for itself,"[52] highlighting the fact that, unlike natural languages, fluency in programming language seems impossible to achieve; those who learn a programming language never truly "go native."

There are many different types of programming language currently in use, causing people to wonder why. Much of the reason is pragmatic: some languages are easier to use for some types of control (screen control, large batch control, etc).[53] Importantly though, "just as natural

languages constrain exposition and discourse, so programming languages constrain what can and cannot be expressed, and have both profound and subtle influence over what the programmer can *think*."[54] Sherry Turkle agrees, asserting that "different computer languages and architectures suggested different ways of thinking,"[55] which is consistent with how the comments are reflective of the communities that use them. The language and the architecture may be used in various settings, so the setting and the language co-create the community as actors in the wider programming network. Individual programmers put their influence on the thoughts that get programmed into the computer, but those people are inherently directed to some degree by the language used and the metaphors implied in those languages. So what is going on is both technological and cultural.

From a purely technological standpoint, the language (and, to a large degree the environment around the language) determines what can be thought and programmed, but the culture that is built up around the language also limits what can be thought of and executed within that language. COBOL has no cut and paste libraries built into the language and cannot be used to create web pages, requiring a "front-end" in a web-friendly language like Java or C#. Therefore, these things are technically impossible in COBOL. Could these capabilities be added? Perhaps, though the difficulty level of the task might be cost-prohibitive, the real problem is that the communities built-up around COBOL are not structured in a way that makes such tasks important. These tasks are culturally impossible to think in COBOL, reinforcing the technical impossibility—in Turkle's sense, the computer determines the thinking that is possible.[56]

The notion that language constrains the possibilities for thought gets at the heart of the combination of the expressive power of the programming language and its syntax with the

expressive power of natural language. Natural language is included in the comments because it is more expressive in some ways.

Therefore, I suggest it is the lack of native fluency that helps give rise to the use of comments. The programming languages are not expressive enough for all tasks, but even those concepts that can be communicated would not necessarily be understood by all the readers.

Further, programming languages are viewed metaphorically, so natural language is given a position of superiority as a communications device, even when symbolic logic is actually superior with regard to a specific task. In the comments found within the source code, one can find examples of complex problems explained in natural language, when the concept is simpler in symbolic logic.[57] There are also examples of explanations that are unnecessary by the pure redundancy with the code that it seeks to explain.

```
/*
 *  encode an unsigned long into a comp_t
 *
 *  This routine has been adopted from the encode_comp_t() function in
 *  the kern_acct.c file of the FreeBSD operating system. The encoding
 *  is a 13-bit fraction with a 3-bit (base 8) exponent.
 */

#define MANTSIZE  13                      /* 13 bit mantissa. */
#define EXPSIZE   3                       /* Base 8 (3 bit) exponent. */
#define MAXFRACT ((1 << MANTSIZE) - 1)    /* Maximum fractional value. */
```
[58]

The above example displays the belief that English is a better communications mechanism than the C programming language. The writer considers it necessary to explain items that are self-explanatory. If you know the language, “MANTSIZE,” “EXPSIZE,” and “MAXFRACT” are clear, being both a part of the language C and part of the language of mathematics, yet they are explained in the comments to the right. The paragraph explanation above the code also goes over the same information. The reader does not truly need both, but the

author was uncomfortable with just one. However, the whole example points back to the inability of a reader to go native; the conceptual universe is not considered rich enough for even mathematical expressions to be clear.

The Corporate sample reflects similar biases toward "natural language" explanations in situations of high complexity regarding processing many variables and conditional processing (that is, to use "if" processing, as in, for example, saying "if this is Tuesday, then this must be Belgium", to borrow a phrase). In the example below, the conditional processing is so complicated that the paragraph in English is almost hopelessly complex, using eleven (11) lines of text to explain a passage of code with only thirty (30) lines. While, to me, this clearly supports the notion of a culturally programmed assumption that an English explanation will be inherently more expressive than code, even when the code is clear and/or the algorithm is so complicated that symbolic explanation is likely more appropriate, another possibility might be that, given a sufficiently complicated algorithm, a developer is expected to provide clarification, regardless of their ability to do so:

```
05273 ****************************************************************
05274 *   TO PERMIT THE USER TO CHANGE ONLY THOSE ELEMENTS OF THE ORIG
05275 *     PHONE WHICH ARE IN ERROR, THE ORIG PH IS HELD IN A FIELD
05276 *     CALLED HOLD-ORIG-PHONE.  MODIFIED ELEMENTS OF THE ORIG
05277 *     PHONE ARE MOVED FROM THE MAP AREA TO THIS COMMAREA. EDIT
05278 *     EDIT CHECKS ARE THEN PERFORMED ON HOLD-ORIG-PHONE. IF THE
05279 *     IF THE EDIT IS PASSED, IT IS DETERMINED IF THE NEW ORIG
05280 *     PH CAN BE SERVICED BY THE CUST'S ACCESS#. THIS CHECK IS
05281 *     DONE BY COMPARING THE CUST'S PRESENT ORIG CITY CODE
05282 *     TO THE OCC OF THE NEW ORIG PH. IF THE 2 OCCS ARE NOT =
05283 *     A WARNING MSG IS DISPLAYED.  THE USER CAN OVERRIDE THIS MSG
05284 *     & PROCESS THE ORIG PH # CHG BY PRESSING 'ENTER' AGAIN.
05285 ****************************************************************
```

[59].

Despite lengthy arguments about the superiority of a given programming language for a particular task, programmers fall back on the natural language communication afforded by

comments. The result is that each executable text becomes a blend of multiple language spaces,[60] bringing together natural language (usually English), technical sub-languages / jargon (usually mathematical, but including business contextual information), and the language used for the symbolic processing itself.

Finally, while programming languages are metaphorical, the metaphors also apply in the other direction, with the language itself becoming a metaphor for the communities that use them, proving that languages have yet to "retreat to the background"[61] and remain a central part of the programming discourse. In some senses, arguments over language and semantics are both the root of the discourse and a smokescreen around other substantive issues (or lack thereof). The programming language becomes a metaphor for the community that uses the language, as in "C programmers are just so arcane and have very little grasp of interpersonal relations," or "UNIX programmers just cannot GREP the solution," or "BASIC programmers are just a series of GOTO statements," or "COBOL programmers are just a series of MOVE statements," and many other statements that have formed a part of my own personal experience.[62]

There are other ways that languages are used as metaphors that help establish community. Each language "entails different styles of programming and suggests different modes for conceptualization,"[63] and may also suggest entirely different ways of forming communities around different metaphors embodied in the language(s) used in that community, with a language's cultural implications, based in the technology and syntax of the language, and sometimes even the name itself.

From the technological perspective, C is considered a "lower-level" language than COBOL. Higher-level languages "abstract" many of the functions of the machine, meaning that designers of the language attempt to "hide" complexity from the individual programmer, to make

programming easier, faster, and less prone to the creation of bugs. As an example of "abstraction", specific memory spaces, for example, are not addressed directly in COBOL. Instead, COBOL allows a programmer to simply define a variable by name and the environment takes care of ensuring that the variable is whole and accessible within certain parameters (e.g., the operation of the program that created it). C gives the programmer more power over items at the hardware level. Programmers can allocate variables to specific memory addresses, allowing faster and more flexible access by other processes. However, directly assigning locations means that a programmer can over-write other information that may have already been stored in that location, potentially causing many programs to fail.

The name COBOL stands for "Common Ordinary Business-Oriented Language" and the language is often thought of as "common" and "business-oriented," representing "the uniformity of mass culture that buries the individual in the crowd."[64] These metaphors, which are embedded in the name of the language, influence the types of communities that will arise around the language. Similarly, C is arcane in both structure and name. Java, one of the most dominant languages at this time, is named in a way that appears to rely on the metaphor of the programmer up all night—the cowboy or the hero.

Turkle further highlights the importance of languages to the self-definition of programming communities, when she relates the tale of "Software Wars." Software Wars is a parody of *Star Wars*, with the moral struggle between forces of conformity (represented by languages like PASCAL and COBOL) and freedom (represented by the hackers and their language of choice, LISP).[65] This idea clearly suffuses programming cultures at some level, but likely more in the FLOSS cultures that more fully embrace the notions of freedom, individuality, and (ironically) community.

While important as foundational elements of programmer culture, these metaphors are not guaranteed to establish community and are not necessarily stable supports for discourse, though they do still point to key themes in programmer identity. Because the names of languages are relatively stable and generally received by the large portion of a community, comments remain the primary location where individual programmers can actively contribute to the construction of community.

Mapping Comments: Form/Function Grid

Network interoperability conferences have been said to "highlight the performative nature of standards. As public spectacles, these events implicitly asserted the importance of standardization itself."[66] Similarly, comments have standards or norms of form and function (that is, the passing along of knowledge about the program), but they also serve a performative or identity-oriented function in both the Linux and Corporate environments. In each, the individual writes for the consumption of the group, staging his/her own performance with each comment (and each line of code, for that matter) and how the individual chooses to stage that performance reflects on the group dynamic in which that performance occurs.

In order to better visualize the many roles a single comment may play within an executable text, I have conceptualized each comment as consisting of both form and function, which I have then mapped on a grid (see Figure 2).

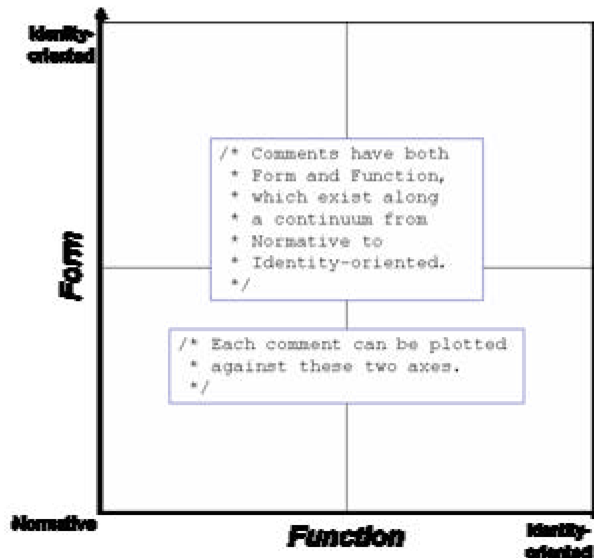


Figure 2: Form & Function Framework for code comments

The grid is in the form of an x-y axis, where the origin represents the normative form and normative function. Traveling to the right, along the x-axis, the function becomes increasingly identity-oriented. Traveling up, along the y-axis, the form becomes increasingly identity-oriented. Hence, a completely normative comment (in both form and function) would be plotted at or adjacent to the origin, while a completely identity-oriented comment would be plotted in the far upper-right corner.

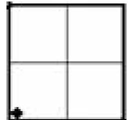
To help visualize this organizational scheme, I will include small icons with each subsequent code snippet to orient the reader with my categorization for this particular snippet.

Icons representing Linux will appear in the right margin, while icons representing the Corporate sample will appear in the left margin.

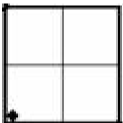
The two comments discussed above, showing the supremacy of natural language over the programming language, are normative comments, regardless of their appropriateness or clarity, since they factually relate the information in the code to which they refer. The mantissa, exponent, maximum fraction example from the Linux sample would be represented this way:

```
/*
 *  encode an unsigned long into a comp_t
 *
 *  This routine has been adopted from the encode_comp_t() function in
 *  the kern_acct.c file of the FreeBSD operating system. The encoding
 *  is a 13-bit fraction with a 3-bit (base 8) exponent.
 */

#define MANTSIZE  13                      /* 13 bit mantissa. */
#define EXPSIZE   3                       /* Base 8 (3 bit) exponent. */
#define MAXFRACT ((1 << MANTSIZE) - 1)    /* Maximum fractional value. */
```
[67]

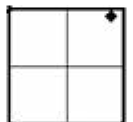


And the complex explanation from the Corporate sample would be similarly represented this way:



```
05273 *****************************************************************
05274 *   TO PERMIT THE USER TO CHANGE ONLY THOSE ELEMENTS OF THE ORIG
05275 *     PHONE WHICH ARE IN ERROR, THE ORIG PH IS HELD IN A FIELD
05276 *     CALLED HOLD-ORIG-PHONE.  MODIFIED ELEMENTS OF THE ORIG
05277 *     PHONE ARE MOVED FROM THE MAP AREA TO THIS COMMAREA. EDIT
05278 *     EDIT CHECKS ARE THEN PERFORMED ON HOLD-ORIG-PHONE. IF THE
05279 *     IF THE EDIT IS PASSED, IT IS DETERMINED IF THE NEW ORIG
05280 *     PH CAN BE SERVICED BY THE CUST'S ACCESS#. THIS CHECK IS
05281 *     DONE BY COMPARING THE CUST'S PRESENT ORIG CITY CODE
05282 *     TO THE OCC OF THE NEW ORIG PH. IF THE 2 OCCS ARE NOT =
05283 *     A WARNING MSG IS DISPLAYED.  THE USER CAN OVERRIDE THIS MSG
05284 *     & PROCESS THE ORIG PH # CHG BY PRESSING 'ENTER' AGAIN.
05285 *****************************************************************
```
[68].

Rarely is a comment completely devoid of both normative form and function, but some come very close, as in the following example.



```
*   "The futexes are also cursed."
*   "But they come in a choice of three flavours!"
```
[69]

The futex comment conveys functional information only in as much as the reader is intended to understand that a futex is a complicated item that can be tricky to use. Instead, the reader is presented with arcane and ironic humor, from which it might be easier to establish that the author was not American, given the spelling of "flavour". This comment is best plotted in the upper right, representing the most fully identity-oriented form and function, since the information relevant to the program that is conveyed is done so in a highly ironic, joking way that requires some sophistication on the part of the reader.

I must emphasize that that Form/Function Grid is intended as a tool for visualization. Mapping the form and function against normative and identity-oriented purposes is not sufficient to capture the subtlety of meaning and use in the comments. A comment that follows the norms of the community may fail to be useful because it includes too much information, or perhaps not enough. Conversely, a comment that seems completely identity-oriented may provide critical information about the program (or the wider environment) that could not easily be gleaned in other ways. More subtly, no comment is ever devoid of group identity, since the basic form is determined by the language, environment, and culture in which the comment is written. What seems purely normative in one setting seems highly identity-oriented in another. This simply highlights the wider community-related identity work performed by the comments and their structure, down to the presentation on the screen.

"Good" Comments

How is it possible to identify a "good" comment? Looking at the entire context, one writer put it best, saying, "A good program is necessarily a well-documented one."[70] In particular, the point is to ease later maintenance, as one cognitive theorist notes, "only when the program is to be modified in some way does documentation become important."[71]

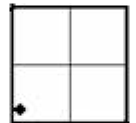
Looking directly at the usefulness of specific comments, a theorist concludes that his "theory predicts that comments which precede a group of statements, and which describe them in terms of operations in another domain, will be particularly helpful."[72] This same theorist specifically asserts that "the availability of prose explanation of the algorithm will have a much larger influence on the speed with which the programmer understands the program than variations in the structure of the program."[73]

Concrete directions regarding comments are not easy to find, but one of the most direct is from Richard Stallman at the GNU open-source project.[74] Stallman requests authors to "put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for. […] If there is anything nonstandard about its use […], or any possible values that would not work the way one would expect […], be sure to say so."[75] This is "good" documentation, meaning highly normative, supporting the stated norms and practices of the programming profession.

The following is a series from a single file in the Linux kernel. This particular file controls interactions with the CPU (central processing unit, which is the main chip that runs the computer). These are "good" comments both individually and collectively. The series appears as individual comments in specific locations throughout the 238-line file and are presented in order of their appearance from top to bottom of the file.

The first comment is an example of the practice of denoting that a particular "if" statement is being closed, much as Richard Stallman recommends in the GNU Coding Standards. The second comment clarifies, though somewhat redundantly, what is happening in a variable declaration. The third comment clarifies that the command is actually actionable and is the point in the program where the CPU is taken off line from the particular process. The fourth comment highlights the fact that a particular line of code, which might be considered out of place or inelegant, is intentionally included to handle specific hardware configurations that were particularly difficult. Finally, the last comment is used to clarify what the purpose of the call is, noting that the CPU is being prepared for further activity, or brought back on line.

```
EXPORT_SYMBOL_GPL(lock_cpu_hotplug_interruptible);
#endif      /* CONFIG_HOTPLUG_CPU */
[… ed.: many lines of code removed …]
/* Take this CPU down. */
static int take_cpu_down(void *unused)
[… ed.: lines of code removed …]
      /* This actually kills the CPU. */
      __cpu_die(cpu);
[… ed.: many lines of code removed …]
      /* Arch-specific enabling code. */
      ret = __cpu_up(cpu);
      if (ret != 0)
            goto out_notify;
      if (!cpu_online(cpu))
            BUG();
[… ed.: many lines of code removed …]
      /* Now call notifier in preparation. */
      notifier_call_chain(&cpu_chain, CPU_ONLINE, hcpu);[76]
```

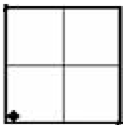


Individually, the comments offer factual clarification of each particular line or series of lines of code with which they are paired, making them firmly normative in function. In addition to their status as individual examples of "good" comments, the four can be viewed collectively. As a group, they highlight the overall function of the program, showing how a CPU can be interrupted, turned off, and subsequently prepared to be turned back on. The series becomes a parallel narrative running alongside the program code. Also worthy of note is the fact that a

single narrative of this kind is generally not the only narrative at work within a given program of significant complexity.

It is also important to acknowledge the use of metaphors in the comments and programming in general. While I have classified the information as "purely factual," and thus normative, the program speaks of "killing" the CPU and the command is executed as a directive to "die." These metaphors are a standard part of programming and are not considered novel or in anyway problematic by the practitioners. In this way, even "good" comments reflect something about the culture in which they are written, including the wider culture of programming as a whole.

In the Corporate sample, the community values are different, making for different choices about comments, particularly in the inclusion of much obsolete code that has been commented out. There are, conversely, few examples where an edit had been performed and subsequently, the same code removed. In the next example, instead of leaving the code, the editor placed a comment marking and explaining what was done, which is extremely useful to history.



```
01827 *PSR3
01828 *      WHEN SUB PROMO IS PASSED, SET UP PGM TO UPDATE FF INFO IF
01829 *      SUBSEQUENT EDITS ARE COMPLETED ERROR FREE
01830 *          LOGIC TO UPDATE WAS PREVIOUS LOCATED HERE
01831 *          IT HAS BEEN MOVED TO THE 0840 PARAGRAPH
01832 *
```
[77]
.

The above example conforms to the GNU expectation of proper documentation, where their guide says flatly: "Don't keep commented out code. Just remove it or add a real comment describing what it used to do and why it is changed to the current implementation."[78]

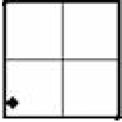
Where "good" comments begin to occupy other spaces is in the descriptive content. Discussing maintenance, one writer noted that a developer, "having exhausted the resources of

local myth and legend, has no alternative but to actually read the code he is required to fix."[79] However, the comments often rely on just this local myth and legend as part of the explanation process, as the inclusion of project names (for example, "METEOR," which appears in the Corporate sample) in the comments indicates. In fact, myth and legend are key elements for readability in the comments.

But if the primary or "official" goal of a comment is to communicate the functions of the program that are notable or unusual, there are likely counter-norms—"good" comments gone "bad." These are essentially "good" comments in the sense that they convey information that is relevant to the program at hand, lending explanatory assistance, where needed. However, that explanation, however well-intentioned, can become a hindrance if it requires too much effort to be kept in synch, or if it conveys merely a redundant version of what can easily be seen in the symbolic code itself.

The code in the next example is relatively clear, so there is some question on the use of extra explanation. The code is designed to return an error to the screen, when the user presses any function key (called "PF" keys on the mainframe). Instead, the user is required to use the "enter" key to process their transaction, hence, not a single function key is valid. Somewhat understandably, the code must create the error message by testing each PF key. A truly "good" comment would probably summarize the logic and say something more like "if any function key is pressed, send 'invalid action key' message to screen" because this would allow a developer to quickly see what is happening. Instead, the example lists out every PF key name (there are only 24), concluding by giving the message "invalid action key" immediately before providing exactly the same information in symbolic form:

```
02773 *----------------------------------------------------------------*
02774 *-   IF PF1, PF2, PF3, PF4, PF5, PF6, PF7, PF8, PF9, PF10
02775 *-   IF PF11 PF12, PF13, PF14, PF15, PF16, PF17, PF18, PF19, PF20
02776 *-   IF PF21 PF22, PF23, PF24, SEND MSG 'INVALID ACTION KEY'
02777 *----------------------------------------------------------------*
02778      IF EIBAID = DFHPF1 OR DFHPF2 OR DFHPF3 OR DFHPF4 OR DFHPF5
02779                  OR DFHPF6 OR DFHPF7 OR DFHPF8 OR DFHPF9
02780                  OR DFHPF10 OR DFHPF11 OR DFHPF12 OR DFHPF13
02781                  OR DFHPF14 OR DFHPF15 OR DFHPF16 OR DFHPF17
02782                  OR DFHPF18 OR DFHPF19 OR DFHPF20 OR DFHPF21
02783                  OR DFHPF22 OR DFHPF23 OR DFHPF24
02784          MOVE 'INVALID ACTION KEY, PLEASE USE ENTER TO PROCESS'
02785                  TO ERRORO
02786          MOVE -1 TO F8NPAL
02787          PERFORM 9000-SEND-DATAONLY-RETURN
02788             THRU 9000-SEND-DATAONLY-RETURN-EXIT.
```



[80]

Visually, this is a complicated comment because the reader must count up all the items listed in the comment in order to figure out if any are missing instead of having a short-hand. At that point, reading the code would present the same information significantly lower likelihood that the code would be out of synch with the requested function of the screen. The comment attempts to present the same knowledge to the reader in roughly the same format, though only the code receives any outside validation.

In another case, a Corporate developer attempts to document an entire process with a 49-line comment, formatted as a table.[81] This comment goes beyond the usual level of documentation.[82] Interestingly, the same edits were again highlighted in yet another 17-line comment just a few lines down the program.[83] Put more bluntly by one author, “We need to warn that although comments are often very helpful, they are subject to being vague, misleading, or even wrong.”[84]

Unsigned inline comments can be vague in a different sense. If these comments appear highly descriptive, they might seem to be written by the original author. In the case of one program in the Corporate sample, where the comments were indeed written the by original

author,[85] the comments have a high likelihood of accuracy and appear "good" in the classic sense,[86] being verbose and descriptive, explaining the purpose of each operation and the ramifications of failure from a programmatic and business perspective. As comments by the original author, the comments are assumed to be more reliable or at least more relevant. This may present a problem in later years when edits have been performed, but the comments remain unsigned, lending some undeserved credibility to the content, potentially leading people astray.

The writers at the Linux Information Project acknowledge that bad documentation is common, citing many problems, including "incompleteness, lack of clarity, inaccuracy, [and] obsolescence."[87] One writer acknowledges that comments, no matter how well-intentioned, can contain suspect information, saying, "The comments do not affect the meaning of the text source but they may help readers to discover the intended meaning,"[88] with an emphasis on "may." Despite problems associated with comments, writers continue to believe in their importance, causing some to seek automation, with one trying "to construct an interactive tool which helps programmers in the documentation step of the programming process,"[89] though this does not solve the problem of meaning in the comments that result.

The primary problem with comments, good or bad, automated or manual, is that information placed in a comment, like the 49-line table-formatted comment of the PF key listing mentioned above,[90] is not required to be maintained by any enforcement mechanism, unlike the code itself. With the code, a compiler enforces at least minimal syntactic correctness because a document that violates the grammar of symbolic logic will fail to compile. Also important is the fact that the compiled code will generally be tested and used. Testers and users will have standards for the functions that are supposed to be supported by the software. If the PF keys, as an example, allowed some processing to occur on the screen mentioned above, a tester would

have the opportunity to list that as a defect or bug. Alternatively, a user might report unexpected results to the developers.

Since comments are ignored by the compiler and therefore cannot be tested by a testing process or by the users themselves, the only reason a comment will stay in synch with the logic is if the developers believe it is important to their own understanding. This problem of code and comment synchronization is a source of concern for theorists, based on the seemingly contradictory stances they project regarding the use of comments. One classic book on programming style asserts, "The only reliable documentation of a computer program is the code itself. [...] Only by reading the code can the programmer know for sure what the program does."[91] The same book goes on to relate the characteristics of good code, saying, "The best documentation for a computer program is a clean structure. It also helps if the code is well formatted, with good mnemonic identifiers, labels, and a smattering of enlightening comments. Flowcharts and program descriptions are of secondary importance."[92] These computer scientists conclude by reinforcing the inherent disconnection between the two spheres of knowledge represented by the program and the comments, writing, "Whenever there are multiple representations of a program, the chance for discrepancy exists. If the code is in error, artistic flowcharts and detailed comments are to no avail."[93] However, the same authors go on to devote thirteen pages "to style in commenting,"[94] further stating, "[...] An excellent program [...] is thoroughly commented and neatly formatted."[95] Clearly, comments are a critical element of good programming. A different theorist more strongly emphasizes the usefulness of comments, saying, "The availability of prose explanation of the algorithm will have a much larger influence on the speed with which the programmer understands the program than variations in the structure

of the program."[96] It is this latter standpoint that is dominant in the programming industry, making commenting an assumed part of the normative programming practice.

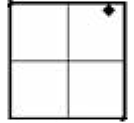
Identity-Orientation

Of course, that programmers include comments at all is a social convention, as much practice as knowledge. Social conventions cover a range of purposes that far outweigh the official goals of comments within programs ("good" comments). Social conventions surround the very structure of the language in the comments, which is often used to further particular social or psychological goals, though these efforts are different in each sub-community of programmers.

One satirical resource on comments asks, "Why should you stop and explain?," pointing out that "Time spent explaining, documenting, commenting — dude! — that's time you could be using to crank out yet more mind-altering code."[97] On one hand, this satire confirms the opinion that programmers are just not good at writing comments.[98] On the other, the satire moves on to confirm how the comments that get written strongly reflect programmer group and personal identity.

The satire presents an imaginary tool to "automatically" insert comments into code. They present the reader with a simple three-line Java code snippet, saying, "Consider the following code fragment, and watch how The Commentator transforms these mystical incantations into readable, well documented source, perfectly tailored to your personality," with various settings for attributes of the text, including "relevance," "humor," "verbosity," "bitterness," "profanity," "FUD" (Fear, Uncertainty, and Doubt), "self-importance," and "religious reference."[99] To the code snippet, they add:

```
/*****************************************************
* okay, finally we are ready to take the important step of
* summing the integer elements of a. I've researched the
* most efficient algorithm and settled on this one,
* presented by Knuth. I don't quite agree with his
* reasoning but the algorithm is sound (did I tell you
* about the cheque I got from Knuth? no? It was back
* in my uni days when I was writing my thesis (youngest
* ever accepted into the program) in TeX on the PDP-10.
* I just couldn't get it to format my differential
* equations properly, and a quick look under the hood...
```
[100]

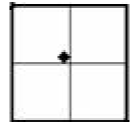


With "verbosity" and "self-importance" set to "10." Note that even this made up comment provide some factual information, telling the reader that the "integer elements of a" are going to be summed at this point in the make-believe program, so there remains some, albeit slight, normative function and form in this comment (far less of the latter). This satire highlights the difficulty in finding a comment with absolutely no normative content; it seems as though programmers resist the temptation to use comments for completely identity-oriented purposes, choosing to base their identity-orientation on some thread of factual relevance.

From a completely serious perspective, some elements of the open-source community cite two basic reasons for bad documentation: 1) to "keep certain aspects of the software secret, for example, undisclosed techniques for developing applications for the software that might only be shared with a few favored developers" and 2) "to facilitate having a lucrative business of selling expensive service contracts and consulting services for the software."[101] These reasons can apply equally well within a single company or community. Comments are a way to project power, knowledge, and control, either directly through the comments themselves, or indirectly through purposefully missing or obtuse comments meant to render a particular developer essential or result in reverence for him/her in a wider community of developers.

Despite the humor content of the futex example above, a conversation seems like an even clearer way to emphasize the social dimensions of comments, since a conversation shows that the medium is not a static history of changes to the program or a "prose explanation of an algorithm"; this is a living document that is constantly being recreated.

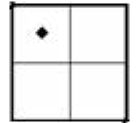
```
/*
 * This needs some heavy checking ...
 * I just haven't the stomach for it. I also don't fully
 * understand sessions/pgrp etc. Let somebody who does explain it.
 *
 * OK, I think I have the protection semantics right.... this is really
 * only important on a multi-user system anyway, to make sure one user
 * can't send a signal to a process owned by another.  -TYT, 12/12/91
 *
 * Auch. Had to add the 'did_exec' flag to conform completely to POSIX.
 * LBT 04.03.94
 */
```
[102]



The above comment from the Linux kernel is more than just for ease of maintenance; it notes an area that might have bugs. The author of the first paragraph admits to lacking understanding. The second paragraph appears to be written by someone else (or, at the very least, the same author at a much later time, responding to additional information), creating a conversation almost as if the code were truly alive and the conversation happening in real-time. The last paragraph is definitely another author who seems to have begrudgingly added code for POSIX compliance and was not happy about it, judging by the "auch." Note that the conversations were considered important enough to the community that none of the paragraphs were removed by later editors, creating an almost exegetical status for the document, with successive editors expanding the analysis. Also, the function of these lines is flexible, as they discuss the code, but focus on the approaches to it and varying interpretations. More dominant here is the form, which relies on both conversation and much personal information, as in the "auch" comment, placing this example in the upper half of the form/function grid.

In another instance, the author of a comment sets up a conversation with the wider community, but allows for it to be open-ended, almost a request for help. This comment notes a process that was considered "bad" by the author, but was used anyway, with a simple note to the future, highlighting the need for a fix, when someone had enough time.

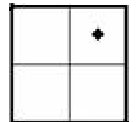
```
/*
 * Select whether the frequency is to be controlled
 * and in which mode (PLL or FLL). Clamp to the operating
 * range. Ugly multiply/divide should be replaced someday.
 */
```
[103]



The comment explains a function of the code, but the ending sentence is more editorial in form, placing this in the upper left quadrant of the form/function grid. Further, the form reinforces the idea that comments are conversations across time, creating both a real and metaphorical discussion in the comments.

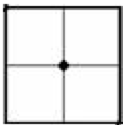
Below is a final example of conversations within the code, but of a much different type; I might call this more of a plea for help.

```
if (IS_ERR(p)) { /* Should never happen since we send PATH_MAX */
      /* FIXME: can we save some information here? */
      audit_log_format(ab, "<too long>");
} else
      audit_log_untrustedstring(ab, p);
kfree(path);
}
```
[104]



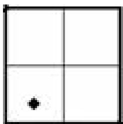
The situation may also be seen as another link between languages; there is no direct way in a programming language to ask for help. The request requires the author to return to natural language. Such a structure offers little explanation of the symbolic code, functioning explicitly as a marker for other human editors, and the form is a direct plea, placing this comment in the upper right quadrant of the form/function grid.

The Corporate sample also contains much “discussion”, though it falls generally in the latter style of unanswered requests or questions. One common example is indeterminacy regarding program or business function. The example below is not a rhetorical question, it is an actual question expressing a lack of understanding on the part of the author, but we do not know the outcome of the question:



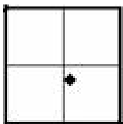
```
08489 *
08490 *  WHAT ARE THE VALID COMBINATIONS OF STANDALONE SERVICES?
08491 *
```
105
.

The next example is slightly more subtle, with the developer noting that some specific processing had to be done with the SBS-CUST-IND, but that was because it “means something,” though what that is remains entirely unclear.



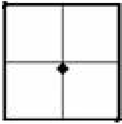
```
10315 * PROD FIX - IF GATEWAY AND CHANGING TO ANY OTHER PAYMENT OPT:
10316 * MOVE SPACES TO FORMER-SBS-CUST-IND - A VALUE OF 'S'
10317 * MEANS SOMETHING TO GATEWAY CUSTOMER LOGIC    2/94 LB
```
106
.

The developer responsible for the next comment apparently decided to remove three lines of code from production, but not without a caveat to future editors:



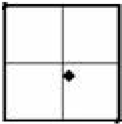
```
01547 * THIS NEEDS TO BE INVESTIGATED...DON'T KNOW WHERE THIS CAME FROM
01548 *    IF ACCT-CANNOT-HAVE-H2H
01549 *       PERFORM 0900-ERROR-PROCESS
01550 *    END-IF
```
107
.

Being unsure of him/herself, the developer in the next comment simply admits it. However, s/he also manages to explain what s/he ended up doing and why, so that future editors might be able to determine “what to do here”:



```
02007 *---NOT REALLY SURE WHAT TO DO HERE - THIS MODULE NEEDED TO BE
02008 *   MODIFIED TO ALLOW THE "OLD" MESSAGE TYPE; NECESSARY, TO ALLOW
02009 *   PLAN UPDATE ON AN ACCOUNT LEVEL DEACTIVATE REQUEST (CALL
02010 *   FROM cobol-7).  THAT BEING SAID, MGHP2-SALES-CITY IS FOUND
02011 *   IN THE PART OF THE HEADER THAT IS NOT INCLUDED IN THE "OLD"
02012 *   MESSAGE TYPE; SO, IN THAT CASE, USE TR1-SALES-CITY INSTEAD.
```
[108].

The next block of code has also been commented out and left with the cryptic statement that s/he "think[s] we need another VoIP IO module here...." It is not clear what that means, but it leaves the door open for a "discussion" between the author and a later editor of the same code:



```
02309 *----THINK WE NEED ANOTHER VOIP IO MODULE HERE...
02310 *    IF  VOIP-SERVICE-PRESENT
02311 *    AND (FINANCIAL-CANCEL-RSN OR LEC-FINANCIAL-CANCEL-RSN)
02312 *        SET TAKEDOWN-VOIP-SVC       TO TRUE
02313 *        SET LG1-REQ-DEACT-VOIP      TO TRUE
02314 *    END-IF.
```
[109].

"We" Construction

There is definitely evidence that comments are formed and have functions outside the normative practices of programming, but there are far more subtle literary forms at work within the comments. Within the Linux kernel, there exists a consistent use of the subject "we" within the comments. This usage performs three functions that help solidify a sense of community among the programmers working on a particular system, be that open-source or proprietary. First, it elevates the image of the group functions. Secondly, it creates boundaries with other groups, particularly users. Finally, it associates the programmers with the machine (in the widest sense, including the software created to run on the hardware).

Notably, these functions need not exist in all communities of programmers. Similarly, the identity-oriented uses of comments within a community may be accomplished with vastly different grammatical constructions, signaling different values and norms within each programming sub-community.

The “we” literary form found in the comments performs three functions. Firstly, the literary form allows elevation of the programmer (and his/her programming discipline) from technician to teacher, associating programming discourse with academic discourse. Secondly, the literary form helps maintain boundaries with outside individuals, organizations, and specifically users. Finally, the literary form expands the group identity to include not just the programmers, but the machines and software systems they create and manage as a cyborg community.[110]

In the Linux kernel, there are 52 files, with 41,505 total lines, including both symbolic code and comments. Out of the total lines, 5711 contained comments (13.75%), of which 738 lines contain the “we” construction (1.78%). Those 738 occurrences appeared in 42 of the 52 files, accounting for more than 80% of the files and represent 12.92% of the total lines of comments. In contrast, the Corporate sample of just seven files contains 28,304 total lines, including both symbolic code and comments. Out of the total Corporate lines, there are 6294 lines of comments (22.24%), of which 36 lines contain the “we” construction (0.13%). These 36 occurrences appeared in five of the seven files and represent just 0.57% of the total lines of comments (see Table 2).[111]

	Linux Sample:	*Corporate Sample*:
Total Files	52	7
Total Lines	41,505	28,304
Lines with Comments	5711	6294
Percentage of Comments to Total Lines	13.75%	22.24%
Lines with “we” construction	738	36
Percentage of Total Lines with “we”	1.78%	0.13%
Percentage of Comment Lines with “we”	12.92%	0.57%

Table 2: Comparison of Comment Distribution

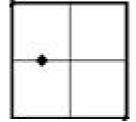
The statistics serve to support several assumptions. First, COBOL programs are often assumed to be much longer than C programs, though this is not a requirement. However, the Linux sample is not even one and a half (1.5) times as large as the Corporate sample is terms of lines of code, despite being more than seven (7) times larger in terms of the number of files. Second is an assumption, also supported by The Commentator, that developers do not like to write comments or explain themselves, believing that good code will simply speak for itself. Despite being larger overall, the Linux sample has 583 fewer lines with comments than the Corporate sample, hinting that developers may comment more when the activity is required. Third, the greater production life of the Corporate sample provides more opportunity for comments to be written. Fourth, and in contrast with conclusion about requiring comments, the practice of commenting out obsolete lines of code results in a much higher percentage of comments to total lines, even though the resulting comments lack explanatory power and may contribute to overall confusion regarding the function of the particular program.

Most importantly, however, the differences between the two samples shed a bright light on the different community values. As the specific examples will support, the lower frequency of the "we" construction within the Corporate sample signifies less emphasis on group identity and boundaries and more emphasis on individual identity through sarcasm and various similar devices. Further, in the Corporate sample, the use of imperative verbs dominate, making the comments with a lecture-like tone take on a more hierarchical sense of an order being issued or perhaps complaints and diatribes against those not "following the rules." Even where the Linux and Corporate samples overlap, in the identity-oriented purposes of the comments, as in group elevation, the comments follow the pattern of group emphasis in the Linux sample and individualism and imperative verbs in the Corporate sample. In this particular case, the Corporate sample also ends up diverging into slightly different intentions.. Perhaps more interesting is where the samples do not overlap, with the Corporate sample displaying little of the boundary work or the cyborg community found in the Linux sample. Correspondingly, the Linux sample displays little of the sarcasm and aggressive challenges found in the Corporate sample.

Group Elevation

In some ways, group elevation is the most subtle, but the most pervasive of the identity-oriented uses of comments. Comments serving this purpose are highly normative in function and yet identity-oriented in form, though often falling along the boundary between the two left quadrants of the form/function grid. The form of these comments gives weight and responsibility to the programmers, making them more than mere technicians.

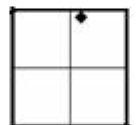
```
/*
 * If we're in an interrupt or softirq, we're done
 * (this also catches softirq-disabled code). We will
 * actually run the softirq once we return from
 * the irq or softirq.
 *
 * Otherwise we wake up ksoftirqd to make sure we
 * schedule the softirq soon.
 */
```
[112]



The form of the above example is that of a lecture in the sense of an academic situation, where the authority figure once typically used “we” extensively. I describe this as a grammar of justification, following Mulkay’s “vocabularies of justification,”[113] where the grammar justifies elevation of programmer status to that of professor or leader.

In some cases, the language of group elevation, including the “we” construction, is strongly supported by the length of the comment. The following example is a comment that begins with a one-line comment that is followed by twenty-eight (28) lines of text describing, “Notes on locking”.

```
/*
 * Check if there exist TIMER_ABSTIME timers to correct.
 *
 * [… ed.: many lines removed …] We are not all that concerned
 * about preemption so we will use a semaphore lock to protect
 * aginst reentry.  This way we will not stall another
 * processor.  [… ed.: many lines removed …]
```
[114]

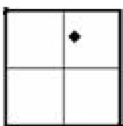


The tone of the selected lines is clearly that of a lecture, as the author expounds on the nature of the locking mechanism chosen and the implications involved in its selection. The author is elevating both his/her own image and the image of the discipline as a whole, creating a striking similarity with the satirical comment included above from the site for “The Commentator”. Taken alone, the first line of this example would be highly normative, but the

lecture outweighs the normative portion 28:1, so that puts the overall form and function in the highly identity-oriented quadrant.

When discussing “good” comments, I noted a distinct conflict over the need to retain inoperable code (that is, code that is commented out). Notably, two of the programs in the Corporate sample have very little code commented out.[115] This lack of code that has been commented out of production (the absence of a comment) is one way to make several conjectures about the nature of the two texts. First, in both cases, the author is the editor, so s/he has no immediate need to retain history. Second, there are likely fewer radical changes, where the old code would be worth keeping as history, even though both programs are over ten years old at this writing. Third, the function is important but does not change radically from year to year. Fourth, the editor being the author, means that s/he does not have a significant problem with indeterminacy—s/he knows what the author intended, since they are one-and-the-same.

Most interesting, however, is one significant block of code that appears to be commented out in one of these two programs:



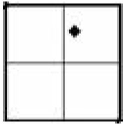
```
00833 *----Note: this non-Registry version, part of a link-to chain,
00834 *    cannot perform a commit work since it affects the calling
00835 *    program as well.  The Registry version can commit...
00836 **** EXEC ADABAS
00837 ****     COMMIT WORK
00838 **** END-EXEC.
```

[116]
.

The appearance of this code seems to invalidate my statement that the program in which it occurs lacks obsolete code that has been commented out. However, this is, in fact, not commented out code in the usual sense, but is actually educational. Looking at the language in the comment that precedes it, we can see that this three-line code snippet was never intended to perform in this program and is here merely to show how another similar program would operate.

This apparently obsolete code is actually a lecture, similar to the “notes on locking” from the Linux sample.

Educational comments appear throughout the Corporate sample, including this one:



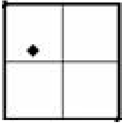
```
00074 *    IMPORTANT NOTE: TO PREVENT INITIALIZATION ISSUES, WHEN ADDING
00075 *                    ANY NEW SUB-PROGRAMS THAT WILL BE CALLED BY
00076 *                    cobol-7 OR ANY OF IT'S SERVICE OR FACILITY
00077 *                    PROGRAMS, THE FOLLOWING IS REQUIRED FOR EACH
00078 *                    MODULE:
00079 *                    1. THE 'INITIAL' STATEMENT MUST FOLLOW THE
00080 *                       PROGRAM-ID NAME.
00081 *                    2. ALL WORKING STORAGE DATA FIELDS MUST
00082 *                       HAVE A VALUE CLAUSE.
```

117

.

The comment above provides developers with basic information about the structure of all COBOL programs. This information is neither specific to the environment (cultural or technical) nor unique to the implementation or methodology; it simply attempts to educate later developers. As an education tool, what this comment shows is factual and accurate, but it clearly can be used to reinforce personal and group identity, with this developer elevating his/her status through an advanced ability to explain COBOL programming and the perception of contributing to proper programming by providing detailed explanations of programming techniques inside the program itself.

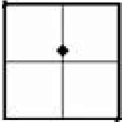
The Corporate sample greatly expands on the theme of group elevation (with more emphasis on personal achievement) by discussing what it means to be a good programmer, often by exemplary commentary as a form of leadership. The developer responsible for the next edit created a begin/end pair to identify his/her change, but did so in a very non-standard way, using three lines for each half of the pair. This seems to indicate a level of importance in his/her mind above the other edits s/he may have performed, showing what a conscientious developer should do in critical situations.



```
13696 ****************************************************************
13697 **START THE COUNTER FOR POSITIONAL SERVICES   name
13698 ****************************************************************
13699              IF  WS-SRV-IDX > 17
13700                  AND         WS-NEW-SERVICE-COUNTER < 7
13701                  AND    (SERVICE-BOTH (WS-SRV-IDX - 1) OR
13702                          SERVICE-RES-ONLY (WS-SRV-IDX - 1))
13703                 ADD 1               TO WS-NEW-SERVICE-COUNTER
13704              END-IF
13705 ****************************************************************
13706 ********END POSITIONAL COUNTER    name      *******************
13707 ****************************************************************
```

[118].

The author of that same begin/end pair also engaged directly in a discussion about the nature of comments themselves. In the next case, the developer points out what information s/he feels is appropriate for the program itself, versus separate documentation. Note the emphatic nature of the comment, indicating how strongly s/he feels about it:



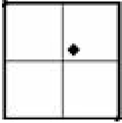
```
00136 * 08/21/98            name         RESTRUCTURE THE PROGRAM TO
00137 *                                  ENHANCE THE ROLLUP LOGIC
00138 *                                  FROM FAC -> SVC -> ACCT.
00139 *                                  !!! PLEASE REFER TO THE LEC
00140 *                                  DISCO DOCUMENT FOR MORE
00141 *                                  DETAILS ON BUSINESS RULES !!!
```

[119].

The discussion about the purpose of comments arises in the contrast between the above example and the comment that follows it in the source code. In the above, the developer insists that the details be stored separately in the project documentation, but the developer who was next to edit the text puts all the details of his/her edit in a 36-line comment that is probably one of the longest change-log comments from the available sample at this company.[120]

Some comments become style guides that travel with the program text. There are several examples in the COBOL source, where comments are consciously written in an attempt to enforce certain conventions. One particular program contained several extended such comments with the following being the most interesting example:

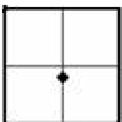
```
02440 *-------------------------------------------------------------
02441 *                      ADABAS CALL ROUTINES
02442 *
02443 *  USE ++INCLUDE SOURCE MEMBERS.  ROUTINE PREFIXES ARE 8000-9699,
02444 *  AS DEFINED BELOW:
02445 *
02446 *            8000 - 8099       table
[... 8 lines of code removed ...]
02455 *            9600 - 9699       MISCELLANEOUS FUNCTIONS
02456 *
02457 * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
02458 *  INSERT COMMENT BEFORE INCLUDE.  EJECT AFTER INCLUDE.
02459 *  CONSULT "BTP SOFTWARE USERS GUIDE" FOR AVAILABLE INCLUDE
02460 *  MEMBERS.  MAKE SURE YOU HAVE NECESSARY COPYBOOKS.
02461 /
02462 *RTR 8/22/94 \/
02463  8010-FINDSET-HOLD-CUSTOMER.
02464 *************************************************************
02465 * ISSUE FINDSET TO HOLD AND READ TABLE DATA
02466 *------------------------------------------------------------
02467 *------------------------------------------------------------
02468 *  ADASQL FOR FIND ON TABLE
02469 *  THIS REPLACES ADAMINT BPFS1200
02470 *  NOTE THAT THE ADAMINT RETRIEVED ALL OF THE FIELDS.  THIS PGM
02471 *  WILL USE JUST A FEW OF THEM.
02472 *------------------------------------------------------------
```



[121].

This "guide" shows the developer how to use the COBOL structure in the current context, with appropriate paragraph ranges defined for each database table and business function. In addition, the comment references knowledge and resources stored outside the program text itself, representing a crossing of "knowledge domains,"[122] to use the phraseology of the industry papers on comprehensibility. This outside reference makes the comment a more active part of the developer discussions. Of course, there is no way to ensure that the document referenced in the comments even exists, but it points to the world outside the program.

In certain cases, the references comments go beyond a mere static reference to an external resource:



```
02459 *******NEEDS ENTRY IN TABLE
```

[123].

This example shows that the developer understands the intersection of knowledge domains. In this case s/he knows that for the proper functioning of the program, an action needs to be taken outside the executable text. The developer chose to highlight that fact for the next editor, since the text is where problem resolution generally starts. The program is the primary shared text—it is used to communicate ideas ***outside*** the program.

While there is a lecturing or educational tone to many of the comments in the Linux sample, there are few, if any, outside references or embedded programming style guides. This disparity between the samples is reflective of the different make-up of each community. Corporate programming groups may (and often do) include members with little or no programming experience, expecting these members to be trained and, importantly, acculturated into the styles and norms of the corporation which hires them. In contrast, programmers in the Linux sample are expected to be proficient in order to be able to contribute, making overt style guides and programming recommendations embedded within comments unnecessary.

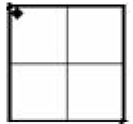
Boundaries

Importantly, the literary device of the "we" construction brings out the boundary work that programmers, like other disciplines, perform on a regular basis, as described by Gieryn.[124] An operating system, to be useful, must have users. If it is widely used, many of those users will not be programmers. These non-programmer users are inherently outsiders, lacking the skills and literacy necessary to be programmers or read code. Despite outsider status, users need to be able to interact with the machine, often at a level that programmers would like to retain for knowledgeable insiders (themselves). The next series of comments deals with the Linux reboot sequence. In the comments are many assumptions about users, their abilities, and their

knowledge. In addition, there are implications about the self-conception of programmers, their importance, and power.

The first comment references “root”. Root users log in to the machine at the “base” of the directory “tree”. Because this log in is at the base of the tree, the root log in can control the entire machine and the experience of other users logging in higher up the tree, hence root is also referred to as “superuser.” Conversely, non-root users are less powerful, having many more restrictions on the functions they can perform and the files that they can view.

```
/*
 * Reboot system call: for obvious reasons only root may call it,
 * and even root needs to set up some magic numbers in the registers
 * so that some mistake won't make this reboot the whole machine.
 * You can also set the meaning of the ctrl-alt-del-key here.
 *
 * reboot doesn't sync: do that yourself before calling this.
 */
asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd, void __user * arg)
{
      char buffer[256];
```
[125]

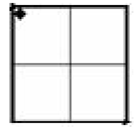


In the comment above, the author states that the reasons that reboot are restricted to root are “obvious”, but that clarity extends only to other programmers and those otherwise on the inside of the boundary. While there are technical reasons the tree structure makes it problematic for a user higher on the tree to execute system-level commands, the implication is that these users are not trusted.

Leaping to conclusions about trust based on one sentence would be problematic, but the author emphasizes this point about trust, when the selection continues a few lines later, noting, “we only trust the superuser” to perform certain functions. The comment also leaves out the contextual information that superusers are generally programmers.

```
/* We only trust the superuser with rebooting the system. */
if (!capable(CAP_SYS_BOOT))
      return -EPERM;

/* For safety, we require "magic" arguments. */
if (magic1 != LINUX_REBOOT_MAGIC1 ||
    (magic2 != LINUX_REBOOT_MAGIC2 &&
                magic2 != LINUX_REBOOT_MAGIC2A &&
            magic2 != LINUX_REBOOT_MAGIC2B &&
                magic2 != LINUX_REBOOT_MAGIC2C))
      return -EINVAL;
```
[126]

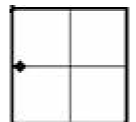


Note also that the keys needed to reboot the entire system are not "keys," or "control characters," or something else. Instead, these are "magic numbers," which constructs a position for programmers as magician or wizard.

The programmers control the magic, through their understanding of the inner workings of the machine and the "mystical incantations" needed to work it, as The Commentator called Java code in the satirical example near the start of the identity-orientation section. Open-source programmers, in particular, tend to be among the elite, so their greater technical understanding may have merit.[127] That this language would imply wizard status for programmers is coherent with the elite group or secret society, since they possess rarified knowledge.

Lest it be said that the boundaries with users are defined only tenuously with the "we" construction, I offer the following example:

```
return -EOPNOTSUPP;     /* aka ENOTSUP in userland for POSIX */
```
[128]

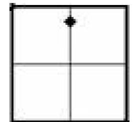


In the above example, the author does not use a personal pronoun at all. Instead the author defines "userland", which is a metaphor evoking geopolitical boundaries, as well as vastly different states of mind (as in "fantasyland", "wackyland", etc). This boundary-oriented

relationship of programmer to user is epitomized by the term “luser,” which combines “loser” with “user”,[129] placing hackers “as holders of esoteric knowledge.”[130]

In a different example that also deals with the reboot sequence, an author writes about a particular function and its relationship to the industry as a whole, establishing boundaries between those on the inside (the contributing programmers) and any other programmers (mainly those who contribute to or support POSIX) who might judge the implementation as deficient.

```
/*
 * setuid() is implemented like SysV with SAVED_IDS
 *
 * Note that SAVED_ID's is deficient in that a setuid root program
 * like sendmail, for example, cannot set its uid to be a normal
 * user and then switch back, because if you're root, setuid() sets
 * the saved uid too.  If you don't like this, blame the bright people
 * in the POSIX committee and/or USG.  Note that the BSD-style setreuid()
 * will allow a root program to temporarily drop privileges and be able to
 * regain them by swapping the real and effective uid.
 */
```
[131]



While the second example does not explicitly use the “we” form, it sets up exactly the same opposition of insider/outsider, through the use of the related “you” construction. This form implies that some readers require education in the ways of the industry and that questioning the programming decisions made in this section is inappropriate and not something done by those on the inside—in the know. In this case, the outsider is a special case, since they can read the code. The outsider is likely a new or potential member of the Linux contributor community.

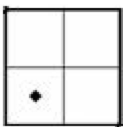
Interestingly, the author also invites the reader to come inside the boundary, as a means to deflect criticism from his/her “deficient” code. The author’s rhetoric can be seen as an attempt to create solidarity between him/herself and the reader (“you”) by speaking to an assumed distaste for an external bureaucratic enemy (“the POSIX committee and/or USG”), who should be blamed for the problem. In a sense, the author invokes political savvy to redeem a

technical shortcoming, using his/her expertise to shift the boundary, potentially deflecting criticism.

These examples of boundary work still provide information about the source code with which they appear, though it could be argued that the informational function is limited. What seems clear, however, is the identity oriented form in each, likely placing them in between the top two quadrants of the form/function grid.

Within the Corporate sample, the boundary work was, surprisingly considering the tone of many comments, much more subtle. In an issue similar to the POSIX controversy in the Linux comment above, the Corporate sample provides evidence of a controversy, where the programmers appear uncomfortable with the result.

The controversy revolves around the appropriate termination to a request to cancel a "service." Before the controversy, the program returned an error when a client requested cancellation of a service that had already been cancelled. Following resolution, the program was changed to "return a false ok" if the service to be cancelled had already reached that state:



```
00184 * 06/03/2003 LOCAL    name          REVISED THE EDITS FOR LOCAL
00185 *            WINBACK                SERVICE DEACT; IF ATTEMPTING
00186 *                                   TO DEACTIVATE AN ALREADY DEACT
00187 *                                   SERVICE, SUPPRESS R1092 ERROR
00188 *                                   AND RETURN A FALSE OK TO THE
00189 *                                   CLINET.
```
[132].

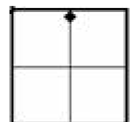
First, note the language of the controversy: the client receives a "false ok." The developers have chosen a linguistic position that challenges the veracity of the program function. Second, comments regarding this controversy total 30 lines, which is 4% of the total comments in this program. Furthermore, every contiguous comment block addressing the "false ok" controversy contains the phrase "false ok," emphasizing the troubling nature of the design. The

implementation is something unusual, so it is worth a comment, but it also significantly changes the operation of the program and what it means to be successful, forcing the developers to voice their concerns in the comments. The rebellion is subtle, but evident. Much more clearly evident is, again, the lack of the community-oriented "we" construction or even the corresponding "you" construction of the POSIX comment from the Linux sample.

Cyborg Community

In the final usage of the "we" construction, the boundaries maintained by the programmers are expanded to include the machine and the software the programmers create to run on it, creating a cyborg community, in the terms of Haraway and others.[133] To be able to extend that boundary, it might be argued that the machine needs to be on the same plane as the programmer. This equalization is not a cognitive leap, since, as one author notes, "animism has become [...] a transparent metaphor, one that is so much a part of the structure of the discourse of the field that we have forgotten that it is there."[134]

```
/* Some compilers disobey section attribute on statics when not
   initialized -- RR */
```
[135]

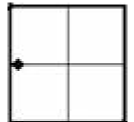


The programmer in the above example notes that the compilers "disobey" some markers, but primarily when not "initialized." Hence, continuing the metaphor, the compilers misbehave when not told what to do. Clearly, the behavior of the compiler is dependent upon the instructions given to it (and how well it was programmed in the first place); the computer has been raised up to a level where the programmer can consider it to be within his/her boundary.

Looking at the compiler comment from Linux sample program softirq.c in terms of the form/function grid, it seems that there is a highly normative function, in that "RR" essentially tells the reader that proper initialization is important. However, the form is highly identity-oriented, since the function is merely implied and the computer so heavily anthropomorphized.

However, the language of the comments moves from merely casting human attributes on the system to an association with the system.

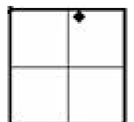
```
/***************************************************/
/* ikconfig_cleanup: clean up our mess           */
```
[136]



This comment seems clear enough, even if the reader does not know what exactly needs cleaning; it is possible to assume that any job leaves some items that need cleaning after completion. However, the form associates the programmer (and the wider community) with his/her system, placing the comment just at the edge of the upper left quadrant of the form/function grid for helping to solidify the notion that the programmer and the system form a cooperative entity.

Revisiting an example used earlier, a very long comment can serve more than one purpose. Above, I showed group elevation in a snippet from a 28-line comment, while another passage in the same comment associates the programmer directly with the machine in the "Notes on locking".

```
/*
 * Check if there exist TIMER_ABSTIME timers to correct.
 *
 * Notes on locking: This code is run in task context with irq
 * on.  We CAN be interrupted!  [… ed.: many lines removed …]
```
[137]



The author does not choose to speak of the CPU or the timer being interrupted, but rather "we can be interrupted". The identity of "we" is very unclear unless the reader allows for the structure of the sentence to define the system and the programmer together as "we".

Sherry Turkle also speaks of anthropomorphization as a tool to deflect blame away from the corporation or programmers who created a system, directing blame instead at the integrated machine/software system itself.[138] In much the same way, programmers, when anthropomorphizing the system have the opportunity to deflect criticism from their own programming or that of others. This technique bears some relation to the reality of creating a complex system.

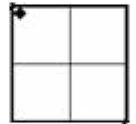
In the Linux sample, the end product is an operating system on which other software will be run, but Linux itself is dependent on even lower-level software. Further, Linux is programmed using other pieces of software that may, or may not, run on Linux itself. The Corporate sample is much the same, relying on many levels of software, including, in this case, an operating system.

When creating these complex systems, bugs are truly instantiations of the union of many people and technologies. There are programmers who create the application at hand, then there are other programmers who create the underlying software that runs the machine and/or the development environment. Then there are programmers responsible for the machine code itself. Then there are the hardware designers, who see their work realized by fabricators, who see their models realized on assembly lines, with technicians, who turn over their products to other technicians to install software for the end user. There are so many pieces involved in the creation of the artifact that is known as the end product of each environment, that the task of tracing a single complex bug to a definite source seems almost impossible.

Hence, deflecting blame through anthropomorphic means may be an effective method of attributing to the system the complexity it deserves. The experienced programmer understands that there are significant limitations to their control of the environment.

However, the association of programmer with the machine remains strong, being most thoroughly realized in this example, which enhances cyborg identity:

```
/*
 * We're trying to get all the cpus to the average_load, so we don't
 * want to push ourselves above the average load, nor do we wish to
 * reduce the max loaded cpu below the average load, as either of these
 * actions would just result in more rebalancing later, and ping-pong
 * tasks around. Thus we look for the minimum possible imbalance.
 * Negative imbalances (*we* are more loaded than anyone else) will
 * be counted as no imbalance for these purposes -- we can't fix that
 * by pulling tasks to us.  Be careful of negative numbers as they'll
 * appear as very large values with unsigned longs.
 */
```
[139]



The literary "we" embedded in this paragraph is powerful in its linking of the author and his/her creation. In particular, notice the parenthetical statement in the third sentence, where "we" technically refers to a particular CPU, but the author clearly associates him/herself with that hardware and embeds him/herself in the software as though directing the function in real-time.

Turkle positions her work as the opportunity to "see the computer as partner in a great diversity of relationships,"[140] but the "we" construction seems to speak to significantly different relation all together. In this case, rather than being a partner, the computer is part of the unit, creating the cyborg relationship, not as partner, but as part and constitutive element. Unlike Turkle's version of programmer anthropormorphizing, the Linux "we" construction does not encourage programmers to think of themselves "like" a machine, where the machine is a model

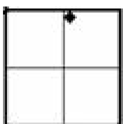
for cognition. Rather, the “we” construction creates the Haraway-like cyborg model, where the programmer is actually part of the machine.

The conceptual plotting of this comment on the form/function grid also highlights an interesting phenomenon. The function is largely informative, explaining a difficult concept that will be critical for later editors, though it could be argued that it goes beyond the normative structures. However, the form, including the use of the “we” metaphor is highly identity-oriented, placing this comment at the top of the y-axis, but likely in the upper left quadrant of the form/function grid, thereby creating a tension between a form that serves identity-oriented goals and a function serving largely normative goals. The complexity of this last example forms a micro case study of how multiple uses and goals for comments can exist within the context of a particular technological frame (or set of frames), without taking any power away from the “official” normative goals.

Corporate Identity

Instead of cyborg identity, the Corporate sample seems to emphasize individual identity using mild humor. In one example, a developer provided commentary on the corporate technology direction with a name for a constant: ADABAS-IS-DEAD,[141] where the constant reflects the fact that the database (ADABAS) is not available, but hints at the industry-wide belief that this particular type of database has no future.

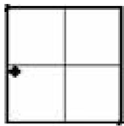
In another situation, an unidentified developer concluded three different, but related edits with the same tag line:



```
04931 * THAT CONCLUDES THE STRUCTURED COBOL PORTION OF THIS SECTION...
04932 * RETURN TO SPAGHETTI CODE!
04933       GO TO 3015-END-OF-CT-EDITS.
```
[142]
.

While the comments are all unsigned, we know the identity of the problem that caused the change, so within the day-to-day workings of the office, the identity of the developer would likely be well known. The humor is also clearly evident, with fun poked at several areas, including the corporate direction on technology, the ability for IT to keep track of user requests for new capabilities, and the programming prowess of prior programmers.

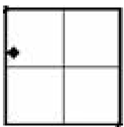
Comments within the Corporate sample also have distinct personal styles, even though they all possess exactly the same textual tools. In some cases, the stylistic differences are subtle, as would be expected in a text-only display, but some developers manage to define a unique style for repetitive tasks:



```
00404 *RTR 8/22/94 \/
00405      05  WS-TRAN-CODE          PIC X(02).
00406 *RTR 8/22/94 /\
```
[143].

The developer identified as RTR completed all inline comments in the same fashion. This is a rather mundane begin/end pairing common to the Corporate sample norms, however, RTR replaced the words “begin” and “end” with the graphical elements defined by paired forward and backwards slashes, in an effort to stamp the entry with his/her personal identity.[144]

Another example highlights the conflict between individuality, visually appealing comments that are visible when scanning a long program, and screen real estate:



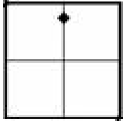
```
02556 *
02557 *>> name    12/19/95
02558 *
02559 *
02560
02561                 MOVE ACCOUNT-ERR TO MGHS1-EXE-RESP-CODE
02562 *
```
[145].

This is another begin/end pairing, but is very different from the others. In this case, the developer decided to use more real estate, opting to make a bigger visual statement. Normally, the pairing (with this level of detail) would occupy only two lines, as in RTR's examples. There are three prominent possibilities for this representational style, 1) the change was of significant importance, warranting greater emphasis, 2) the developer came from a different environment (for example, PC computing), where such presentation styles are more common, or 3) the developer's personal style dictates an emphasis on the grand.

In the final analysis, the change is insignificant at best—a different type of error is moved to the response code—and as such, it does not warrant a more significant highlight in the begin/end pair. The developer at hand is exclusively a mainframe developer, so was accustomed to the reduced real estate common in the mainframe environment.[146] The last possibility comes to the top; the comment is used to subtly communicate his/her personality to the later readers of the program.

Personality comes out best in a final example, which contrasts sharply with the notion that "a good program is necessarily a well-documented one,"[147] and defies the entire community-oriented tone of the Linux sample. In this example, both Knuth and at least one other author would be forced to acknowledge the completeness of the original author's commentary regarding every aspect of the program, with over 28% of the program lines being comments.[148] However, the very first developer to edit the program begins with a jab at the programming abilities of the author:

```
02266 * 06/17/96 name       06/26/96    FIXED FOR PRT#960521.05.
[... 5 lines removed ...]
02272 *                                 ALSO REMOVED THE CODE THAT
02273 *                                 APPEARED TO BE A RESULT OF A
02274 *                                 PROGRAMMER GUESSING HOW COBOL
02275 *                                 WORKED.  THESE GUESSES CAUSED
02276 *                                 WARNINGS WHEN COMPILED,
02277 *                                 ADDED UNECESSARY OVERHEAD AND
02278 *                                 COMPLEXITY TO THE PROGRAM, AND
02279 *                                 OBSCURED THE LOGIC OF THE PGM.
02280 *
```



[149]
.

The comment is not directed at a specific person by name, but the fact that no edits are recorded in the five months between original creation and this edit makes the implication clear, the editor feels the original author is significantly deficient in his/her programming ability. Furthermore, the developer making the edit clearly positions him/herself as an authority of some weight.[150] This is all accomplished with a definitive style, which some might call abrupt.

Conclusion

Comments have an "official" or normative purpose—explain how the program works—and a close reading of various programs in radically different settings shows this to be a valid and "real" use. As Amy Slaton and Janet Abbate write regarding standards, the basic tools of a discipline are perceived as "knowledge rather than practice".[151] Comments are considered to embody the knowledge of a program and even an organization or community. As the items above show, there is indeed knowledge, but the comments are clearly a practice and a performance.

However, that same reading also reveals much more. At the most fundamental level, the structure of comments is strongly influenced by the programming language, the technical environment, and the culture in which they are written. Further, the very existence of the comments highlights the metaphorical status of programming languages. More importantly, while clearly performing normative functions, the form allows metaphorical constructions that help elevate the programming community, establish and defend community boundaries, stretch the definition of programming community to become a cyborg community, including the systems the programmers create and the machines on which they run, and help individuals establish their own identity.

One specific lesson from the comments is that very specific types of community will be created by and reflected in the discussions and interactions in the source code comments. As Sherry Turkle writes, "A computer program is a reflection of its programmer's mind" (Turkle, p. 24), and the comments will occupy a similar mental space. The greater emphasis in the Linux sample on the "we" construction reflects a more collegial atmosphere, highlighting the voluntary nature of the project, which relies on the good will of the collected organization. The Corporate

sample is more combative and directive in tone and structure, reflecting the hierarchical structure of the organizations in which they are constructed.

Importantly, what programmers establish and then reflect through these comments might be called "ownership" or "connection" with the product of the programming endeavor. Speaking of toys, Sherry Turkle says, "When people are asked to care for a computational creature and it thrives under their ministrations, they become attached, feel connection, and sometimes much more,"[152] essentially describing how programmers relate to their work. Following the process of anthropomorphizing, the machines and systems used and built by the programmers become creatures that are nurtured and ministered to by the programmers. In some cases, the creatures are given life by the programmers. This is a feeling common to all programming, not just open source. Any significant amount of time spent caring for a program, or set of programs results in attachment and connection.

It would be tempting to assume that the open source programmers have a greater degree of connection to their projects, but the writing of the programmers in their comments does not seem to support this, with, if anything, more passion expressed by the Corporate programmers than by the Linux programmers. The connection is likely the result of longevity, since both sets of code are relatively old (though the Corporate sample is older), but also specialization. Particular programmers specialize, either by choice, as in the Linux sample, or by a combination of choice and management directive, as in the Corporate example. This helps reinforce the sense of ownership inherent in programming, since, as Sherry Turkle says, "a large computer system is a complicated thing", leaving "plenty of room for territoriality."[153] This territoriality is alternatively to be seen as attachment and connection to the creatures spawned by the care given to the machine.

While some writers have attempted to view comments as explanatory only for the text at hand and to view the texts simply as programs to control a machine, I conclude that comments are a critical resource in the establishment and/or reinforcement of identity on both a group and personal level, through their role in the executable texts. The identity-oriented purposes of executable texts can only be removed from texts when "the theory is unconcerned with personality characteristics of programmers, with the effects of varying motivational conditions or with social interaction in programming groups,"[154] which is a less-than-useful way to approach either a social group or a form of communication.

Pragmatism

Taking a pragmatic stance, a programming team offers a window into their group dynamic and the psychology of specific individuals through its comments. Assailing the prowess another programmer in a comment, as in the last Corporate sample, is not merely an idle jab at the syntax of a passage, it is a personal attack on the targeted programmer and his/her abilities. Watching for markers like these can potentially identify the beginnings of a dysfunctional organization.

Conversely, placing too great an emphasis on stringent comment guidelines is not necessarily productive. Allowing the developers to define their own style will help cement the culture of the community that creates the code. Though some influence over the style of the comments may have impacts on the health of the community that creates them.

Finally, if comments are strongly encouraged with in a specific community, care must be taken to address the currency of those comments. Comments need to be deleted or updated, just like code, or they reduce the comprehensibility of the executable text. While comments can

certainly be retained for historical background, they need to be highlighted as such, though a simple date may be sufficient.

Future Research

I would argue that computer programming is in the midst of a revolution, of sorts, having created the new knowledge economy and spurred the United States economy at the end of the 20th Century, as well as fueling the Indian economy into the 21st Century. In this sense, it is much the same as physics following the end of the Second World War; whereas WWII was the Physicist's War, the turn of the 21st Century might be the Programmer's Economy. Further similarities exist in the age distribution of the two groups. Physics in the post-war period was a very young discipline, with most practitioners being in their early thirties or late twenties. This same age distribution is common in computer programming in current society. Both disciplines viewed themselves as primarily a meritocracy, where the only valid credentials were ability. However, both were also undergoing changes that required a much greater reliance on formalized credentials to establish official gateways into the fields.

A comparison of the internal rhetoric of these two disciplines at these critical junctures, might provide some perspective on possible developmental paths for computer programming, as the field continues to change, based on economic and social pressures.

Additionally, there are many devices used to establish individual and group identity. There is the language, or jargon, used by the group, the literature, film, and art preferred by the group or even considered canonical. All these sources can be brought together with the actual texts written, edited, and annotated by programmers to form a more rich picture of the varying communities of programmers that write the software that runs our world.

Bibliography

Source Archives

COBOL source code. Company confidential. September 1982 – October 1996.

Linux source code. Version 2.6.15, 1/3/2006. Available at: http://www.kernel.org/pub/linux/kernel/, accessed 2/28/2006.

Works Cited

Barnes, Barry. *Interests and the Growth of Knowledge*. Routledge, London, 1977.

"BEA WebLogic Integration Glossary," Version 2.1. unsigned. BEA Systems, Inc., October 2001. http://e-docs.bea.com/wlintegration/v2_1/gloss/glossary.htm, accessed 2006/07/06.

Bijker, Wiebe, E. "The Social Construction of Bakelite: Toward a Theory of Invention" in Bijker, W. E., T.P Hughes, and T.J, Pinch (eds.), *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*, Cambridge, MA: MIT Press, pp. 159-187, 1987.

Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Press, May 1978.

Downey, Gary, Joseph Dumit, & Sarah Williams. "Cyborg Anthropology," in *Cultural Anthropology* 10(2) (1995): p. 264-269.

Edwards, Paul N. *The Closed World: Computers and the Politics of Discourse in Cold War America*. Cambridge: The MIT Press, 1996.

Fauconnier, Giles & Mark Turner. "Conceptual Integration Networks." from Cognitive Science, 22(2) 1998, 133-187. Expanded web version, 10 February 2001.

Gieryn, Thomas. "Boundary-Work and the Demarcation of Science from Non-Science: Strains and Interests in Professional Ideologies of Scientists" in *American Sociological Review* 48, pp. 781-795, 1983.

Green, Roedy. "Unmaintainable Code." Canadian Mind Products, 2006; from: http://mindprod.com/jgloss/unmain.html, accessed 2006/04/14.

Gurtov, Andrei. "Technical Issues of Real-Time Network Simulation on Linux: B.Sc. Thesis." unpublished. Petrozavoksk State University, Russia, Faculty of Mathematics, Department of Computer Science, 29 May 1999. Available at http://www.cs.helsinki.fi/u/gurtov/papers/bs_thesis.pdf, accessed 2006/07/12.

Haraway, Donna Jeane. *Simians, Cyborgs, and Women: The Reinvention of Nature*. Routledge, New York, 1991.

Heiss, Janice J. "Envisioning a New Language: A Conversation With Sun Microsystems' Victoria Livschitz." *Sun Developer Network*. Sun Microsystems, December 2005. Available at http://java.sun.com/developer/technicalArticles/Interviews/livschitz2_qa.html, accessed 2006/07/11.

Kernighan, Brian W. & P.J. Plauger. *The Elements of Programming Style*, 2nd Ed. McGraw-Hill, 1978.

The Jargon File, version 4.4.7. Unsigned. Available online at http://catb.org/jargon/html/L/luser.html, accessed 2006/11/18-14:51et.

Lakhani, Karim R. & Robert G. Wolf. “Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects”. in *Perspectives on Free and Open Source Software*. J. Feller, B. Fitzgerald, S. Hissam, & K.R. Lakhani, eds. MIT Press, 2005. Available at http://opensource.mit.edu/papers/lakhaniwolf.pdf, accessed 2006/10/03-12:50et.

Lakoff, George and Mark Johnson. *Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought*. Basic Books, 1999.

Latour, Bruno & Steve Woolgar. *Laboratory Life: The Construction of Scientific Facts*. Princeton University Press, Princeton, NJ, 1986.

Mehta, Jigar. “Development Inline Comment while working in team.” *The Code Project*; from: http://www.codeproject.com/useritems/inline_comment_macro.asp, accessed 2006/03/01.

Mulkay, Michael J. “Norms and ideology in science,” in *Sociology of Science* 15 (4/5), pp. 637-656, 1976.

Pflueger, Joerg. “Language in Computing.” in *Experimenting in Tongues: Studies in Science and Language*. Matthias Doerries, ed. Stanford University Press, 2002.

Raymond, Eric Steven. “The Cathedral and the Bazaar”. Version 3.0. Thyrsus Enterprises, 2000. Available online at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, accessed 2006/10/17-16:15et.

Rugaber, S. “Program Comprehension”. *Encyclopedia of Computer Science & Technology*, 35, 1996, p. 314-368.

Scott, Michael L. *Programming Language Pragmatics*. Academic Press, 2000.

Semino, Elena, John Heywood, and Mick Short. “Methodological problems in the analysis of metaphors in a corpus of conversations about cancer.” *Journal of Pragmatics* 36 (2004) 1271–1294.

Slaton, Amy and Janet Abbate. “The Hidden Lives of Standards: Technical Prescriptions and the Transformation of Work in America”. In *Technologies of Power: Essays in Honor of Thomas Parke Hughes and Agatha Chipley Hughes*. Michael Thad Allen and Gabrielle Hecht, eds. MIT Press, Cambridge, Massachusetts, 2001.

Stallman, Richard, et al. *GNU Coding Standards*. Free Software Foundation, 8 February 2006; from: http://www.gnu.org/prep/standards/, accessed 2006/02/16.

Stallman, Richard, et al. “Why ‘Free Software’ is better than ‘Open Source’”. Free Software Foundation, 1998. Available online at http://www.gnu.org/philosophy/free-software-for-freedom.html, accessed 2006/10/16-20:00et.

Stallman, Richard, et al. “Why Software Should Be Free”. Free Software Foundation, 24 April 1992. Available online at http://www.gnu.org/philosophy/shouldbefree.html, accessed 2006/10/16-20:00et.

Travers, Michael David. *Programming with Agents: New metaphors for thinking about computation*. Dissertation, MIT, June 1996. Available on line at: http://xenia.media.mit.edu/%7Emt/thesis/mt-thesis.html, accessed: 2006/07/17.

Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005.

“Telecommunications: Glossary of Telecommunications Terms.” unsigned. in *Federal Standard 1037C*. US Government Publication, Friday, 23 August 1996, 00:22:38 MDT. Available at http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm, accessed 2006/07/17.

Weinberg, Gerald M. *The Psychology of Computer Programming*. Litton Educational Publishing, Inc., 1971.

End Notes

[1] Knuth, Donald E. "Literate Programming." *The Computer Journal*, Vol. 27, No. 2. 1984, p. 97.

[2] Knuth, Donald E. "Literate Programming." *The Computer Journal*, Vol. 27, No. 2. 1984, p. 97.

[3] Jones, Duncan Edwards. "The seven secrets of successful programmers." The Code Project; from: http://www.codeproject.com/tips/7secrets.asp, 2006/03/01.
Quoting: "It is one of the most pervasive misunderstandings in computing that the source code is for the computer's consumption. Computers work with low-level binary code, a series of impenetrable 1's and 0's or hexadecimal numbers, not the structured high level languages we code in. The reason that these languages were developed was to help the programmer.
In practice, coding for human consumption often means coding for clarity first, over efficiency and speed second."

[4] "Documentation Definition." unsigned. Linux Information Project, 23 February 2006; from: http://www.bellevuelinux.org/documentation.html, 2006/03/01.

[5] Kernighan, Brian W. & P.J. Plauger, p. 151.

[6] "Documentation Definition." unsigned. Linux Information Project, 23 February 2006; from: http://www.bellevuelinux.org/documentation.html, 2006/03/01.

[7] Knuth, Donald E. "Literate Programming." *The Computer Journal*, Vol. 27, No. 2. 1984, p. 97.

[8] Jones, Duncan Edwards. "The seven secrets of successful programmers." The Code Project; from: http://www.codeproject.com/tips/7secrets.asp, 2006/03/01.

[9] Latour, Bruno & Steve Woolgar. *Laboratory Life: The Construction of Scientific Facts*. Princeton University Press, Princeton, NJ, 1986, p. 45-53, subtitled "Literary Inscription."

[10] Latour, Bruno & Steve Woolgar, p. 53.

[11] Latour, Bruno & Steve Woolgar, p. 53.

[12] Edwards, Paul N. *The Closed World: Computers and the Politics of Discourse in Cold War America*. The MIT Press, Cambridge, Mass., 1996, p. 38.

[13] Edwards, Paul N., p. 38.

[14] Barnes, Barry. *Interests and the Growth of Knowledge*. Routledge, London, 1977, p. 1.

[15] Barnes, Barry, p. 24.

[16] Mulkay, Michael J. "Norms and ideology in science," in *Sociology of Science* 15 (4/5), pp. 637-656, 1976, p. 645.

[17] Gieryn, Thomas. "Boundary-Work and the Demarcation of Science from Non-Science: Strains and Interests in Professional Ideologies of Scientists" in *American Sociological Review* 48, pp. 781-795, 1983, p. 792-93.

[18] Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 287.

[19] Haraway, Donna Jeane. *Simians, Cyborgs, and Women: The Reinvention of Nature*. Routledge, New York, 1991, p. 224.

[20] Haraway, Donna Jeane, p. 203.

[21] My knowledge of the Corporate sample is somewhat personal, coming from a corporation with which I am directly familiar. In some cases, I have direct knowledge of the individual issues and personalities at work in the executable texts. I will sometimes draw on this experience to make the narrative more interesting, but attempt to allow the texts to stand on their own. My contextual experience with the Corporate sample appears most notable in the section describing the various frames, though much of this information addresses common business practices.

[22] This level of control might be more evident when looking at the version of the Linux kernel that I used: 2.6.15, which was further sub-divided into seven minor revision releases (2.6.15.1, 2.6.15.2, etc.), occurring over a three-month timeframe (see: http://www.kernel.org/pub/linux/kernel/, accessed 2/28/2006).

[23] Bijker, Wiebe, E. "The Social Construction of Bakelite: Toward a Theory of Invention" in Bijker, W. E., T.P Hughes, and T.J, Pinch (eds.), *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*, Cambridge, MA: MIT Press, pp. 159-187, 1987, p. 183-84.

[24] Slaton and Abbate, p. 133. Slaton and Abbate speak of network users who consumed hardware, while producing network services. In the case of open source programmers, they are truly both producer and consumer.

[25] This situation might be very different for programmers of proprietary software sold on the open market (e.g., Microsoft Word or Adobe PhotoShop).

[26] This attitude is widely common among programmers and needs no specific support from the literature. I performed a search on monster.com for "C programming" and another for "COBOL programming." The first search returned "more than 1000" hits, while the second returned "632." (www.monster.com, accessed 2006/08/19)

[27] As an example, IBM currently offers Linux utilities that run on its mainframe environments, but those environments still rely on existing mainframe operating systems (for example, IBM's z/OS). see: http://www-03.ibm.com/servers/eserver/zseries/os/linux/, accessed 2006/08/19.

[28] Linux was started in 1991 by Linus Torvalds and the dates on the kernel files reflect this time period.

[29] The term 'development environment' is usually reserved for the programs that offer more extensive assistance. Examples of free, open source environments include EMACS and Eclipse. On the text-editor end of the spectrum is a wide variety of tools that include anything from Crimson Editor (my choice) to MS Notepad, which is standard on MS Windows operating systems.

[30] Several companies contribute to Linux and sell support for a specific version that they distribute. These companies include their identification in the executable texts their employees modify, for example, Red Hat, Inc., and IBM are identified in futex.c, while Hewlett-Packard is identified in configs.c.

[31] Most development projects require a specific piece of software to manage versions (called 'version control'). Usually, this is the free version control software known as CVS. There are other options, so this is a form of centralized control and standardization.

[32] Stallman, Richard, et al., p. 31.

[33] This is consistent with colloquial language. In a 1978 article, Ruven Brooks refers to such comments as "interline comments." Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Press, May 1978, p. 197.

[34] COBOL source code, COBOL-6. October 1996.
Note that the line spacing and presentation of the code snippet samples is important to understanding how they are constructed and viewed by the population for which they are intended. As such, code snippets are inserted into the paper in such a way that they have the same visual appearance that they would when viewed in a program. Hence, they are presented with a fixed-width font, single line-spacing, and margins that are intended to allow each line to

remain intact as it would appear in the program. In particular, the Linux examples use unusual margins to allow for long lines that would otherwise wrap, changing the impression of the snippet.
Similarly, some elements of the program snippets have been changed to assure confidentiality. First, the names of the programs have been changed to "COBOL-N", where "N" is simply a sequential number, assigned mostly in date order (by year only), with the oldest COBOL program being COBOL-1 and the newest being COBOL-7. Second, anywhere in the Corporate sample where a programmer's name appeared as more than initials, that name has been removed. In some cases, the name has been replaced with lower-case letters to signify the change, where it seemed materially important to the presentation. Finally, any potentially proprietary information (e.g., company names) have been removed from the Corporate sample. None of these types of changes have been made with the Linux sample, since the Linux source is open and available.

[35] COBOL source code, COBOL-7. July 1996.

[36] Mehta, Jigar. "Development Inline Comment while working in team." *The Code Project*; from: http://www.codeproject.com/useritems/inline_comment_macro.asp, accessed 2006/03/01.
The author's example in the source document was in a format acceptable for a macro, but when implemented would result in the comment as shown, with the actual date and time replacing the text "[& now() &]."

[37] COBOL source code, COBOL-2. February 1993.

[38] COBOL source code, COBOL-1. September 1982.

[39] Mehta, Jigar.

[40] I have no formal statistics to back up this claim, however, informal statistics are persuasive. In the group responsible for this code, there were five staff members with tenure over 20 years, about another five with tenure between 15 and 20 years, and probably a dozen with tenure between 10 and 15 years. The common cultural perception in the group was that tenure at the company was either exceedingly short (a year or less) or a "life sentence."

[41] In the writings of both Richard Stallman and Eric Raymond, who are two of the primary advocates of FLOSS techniques, open-source is seen as an elite group that is happier, more productive, inherently collective group endorsement – in other words, artists (see: Stallman, Richard, et al. "Why 'Free Software' is better than 'Open Source'". Free Software Foundation, 1998. Available online at http://www.gnu.org/philosophy/free-software-for-freedom.html, accessed 2006/10/16-20:00et; Stallman, Richard, et al. "Why Software Should Be Free". Free Software Foundation, 24 April 1992. Available online at http://www.gnu.org/philosophy/shouldbefree.html, accessed 2006/10/16-20:00et; and Raymond, Eric Steven. "The Cathedral and the Bazaar". Version 3.0. Thyrsus Enterprises, 2000. Available online at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, accessed 2006/10/17-16:15et). Additionally, researchers studying the phenomenon found that "enjoyment-based intrinsic motivation, namely how creative a person feels when working on the project, is the strongest and most pervasive driver. We also find that user need, intellectual stimulation derived from writing code, and improving programming skills are top motivators for project participation" (Lakhani, Karim R. & Robert G. Wolf. "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects". in *Perspectives on Free and Open Source Software*. J. Feller, B. Fitzgerald, S. Hissam, & K.R. Lakhani, eds. MIT Press, 2005. Available at http://opensource.mit.edu/papers/lakhaniwolf.pdf, accessed 2006/10/03-12:50et, p. 3).

[42] Green, Roedy. "Unmaintainable Code." Canadian Mind Products, 2006; from: http://mindprod.com/jgloss/unmain.html, accessed 2006/04/14.

[43] Mehta, Jigar. "Development Inline Comment while working in team." *The Code Project*; from: http://www.codeproject.com/useritems/inline_comment_macro.asp, 2006/03/01.

[44] COBOL source code, COBOL-1. September 1982, lines 08513-08527. The line numbers were removed for clarity in this case, since the intervening lines were also removed.

[45] One common complaint of advanced object-oriented programmers at the same company was that many corporate programs in C++ were "as long as a structured program."

[46] Megill, Norman. "mm.java," 2003; from: http://us.metamath.org/mmsolitaire/mm.java, 2006/02/15.

[47] Semino, Elena, John Heywood, and Mick Short. "Methodological problems in the analysis of metaphors in a corpus of conversations about cancer." Journal of Pragmatics 36 (2004) 1271–1294, see page 1272.

[48] Copybook is a term used in COBOL. "A common piece of source code designed to be copied into many source programs; used mainly in IBM DOS mainframe programming. In mainframe DOS (for example, DOS/VS and DOS/VSE), the copybook was stored as a book in a source library. A library comprised of books, the name of each of which began with a letter prefix designating a programming language (for example, A.name for Assembler, C.name for COBOL) because DOS did not support multiple or private libraries. This term is mainly used by COBOL programmers, but it is supported by most mainframe languages. The IBM OS series did not use the term copybook; instead, it referred to such files as libraries implemented as partitioned data sets or PDSs. A copybook is functionally equivalent to a C or C++ include file."
from: "BEA WebLogic Integration Glossary," Version 2.1. unsigned. BEA Systems, Inc., October 2001. http://e-docs.bea.com/wlintegration/v2_1/gloss/glossary.htm, accessed 2006/07/06.

[49] Gurtov, Andrei. "Technical Issues of Real-Time Network Simulation on Linux: B.Sc. Thesis." unpublished. Petrozavoksk State University, Russia, Faculty of Mathematics, Department of Computer Science, 29 May 1999. Available at http://www.cs.helsinki.fi/u/gurtov/papers/bs_thesis.pdf, accessed 2006/07/12.

[50] "Telecommunications: Glossary of Telecommunications Terms." unsigned. in *Federal Standard 1037C*. US Government Publication, Friday, 23 August 1996, 00:22:38 MDT. Available at http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm, accessed 2006/07/17.

[51] Quoting Alan Perlis in Heiss, Janice J. "Envisioning a New Language: A Conversation With Sun Microsystems' Victoria Livschitz." *Sun Developer Network*. Sun Microsystems, December 2005. Available at http://java.sun.com/developer/technicalArticles/Interviews/livschitz2_qa.html, accessed 2006/07/11.

[52] Kernighan, Brian W. & P.J. Plauger, p. 151. In this passage, the authors imply that extra comments do not help a program that "cannot speak for itself," but I have inverted their statement, implying that comments do much of the 'talking'.

[53] Pflueger, Joerg. "Language in Computing." in *Experimenting in Tongues: Studies in Science and Language*. Matthias Doerries, ed. Stanford University Press, 2002, p. 125 & p. 136.

[54] Scott, Michael L. *Programming Language Pragmatics*. Academic Press, 2000, p. 5, emphasis in original.

[55] Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 9.

[56] In an explanatory footnote, Sherry Turkle further supports the notion that a particular language allows only some kinds of thoughts, writing, "COBOL is designed for manipulations such as sorting records in business applications. Programs of the kind for which the high-level language is specifically designed can be written more quickly and economically in it. But other kinds of programs might be cumbersome or even impossible." (from: Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 168.)

[57] This is a view strongly contrary to that of Lakoff & Johnson (see Lakoff, George and Mark Johnson. *Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought*. Basic Books, 1999, p. 398). The evidence of this phenomenon is stronger in other archives of executable texts.

[58] "acct.c" Linux kernel 2.6.15.

[59] COBOL source code, COBOL-1. September 1982.

[60] Fauconnier, Giles & Mark Turner. “Conceptual Integration Networks.” from Cognitive Science, 22(2) 1998, 133-187. Expanded web version, 10 February 2001.

[61] Pflueger, Joerg, p. 162.

[62] I have personally worked in the IT field, along side programmers using various language and platforms for more than fifteen (15) years. In that time, I have heard statements like this so often that they become an unacknowledged element of the culture.

[63] Pflueger, Joerg, p. 136.

[64] Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 209.

[65] Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 208-11.

[66] Slaton, Amy and Janet Abbate. “The Hidden Lives of Standards: Technical Prescriptions and the Transformation of Work in America.” In *Technologies of Power: Essays in Honor of Thomas Parke Hughes and Agatha Chipley Hughes*. Michael Thad Allen and Gabrielle Hecht, eds. MIT Press, Cambridge, Massachusetts, 2001, p. 135.

[67] “acct.c” Linux kernel 2.6.15.

[68] COBOL source code, COBOL-1. September 1982.

[69] “futex.c” Linux kernel, version 2.6.15.

[70] Deimel, Lionel E., Jr. “The Uses of Program Reading.” *SIGCSE Bulletin*, Vol. 17, No. 2, June 1985, p. 7.

[71] Brooks, Ruven. “Using a Behavioral Theory of Program Comprehension.” Proceedings of the 3rd International Conference on Software Engineering. IEEE Press, May 1978, p. 199.

[72] Brooks, Ruven. “Using a Behavioral Theory of Program Comprehension.” Proceedings of the 3rd International Conference on Software Engineering. IEEE Press, May 1978, p. 198.
This assertion seems a far stretch, based on my analysis of the Corporate sample. The problem likely lies with methodology. Brooks tested his theory with “a set of four FORTRAN programs, ranging in length from 109 to 231 lines,” constructed just for the test. So the test was completely manufactured without allowing time for comments to evolve away from the code and the samples of the program were far too short to be representative of major business programs, as my Corporate sample shows (since COBOL and PASCAL share the attribute of being structured languages that do not shy away from length).

[73] Brooks, Ruven. “Using a Behavioral Theory of Program Comprehension.” Proceedings of the 3rd International Conference on Software Engineering. IEEE Press, May 1978, p. 200.
This assertion seems to conflict directly with another, much more well-cited study. Weinberg cites an IBM sponsored “experiment in which several versions of the same code are produced, one with correct comments, one with one or two incorrect comments and one with perhaps no comments at all” (Okimoto, 1970, in Weinberg, p. 164), concluding that “for certain types of code, at least, correct interpretation of what the program does can be obtained more reliably and faster without any comments at all” (Weinberg, p. 164). This same 1970 IBM study continues to be cited into the second half of the 1990s, with authors using it as a basis to conclude, similar to Weinberg, that “studies of short programs show that comments in code interfere with the process of understanding, [and] if not up to date, can be misleading and cause errors in the semantic representation of the code” (Rugaber).

[74] Also quite explicit is the Sun-sponsored Javadoc document available online (http://java.sun.com/j2se/javadoc/writingdoccomments/index.html, referenced 2006/03/01-12:36) and referenced by several sources, including the GNU Coding Standards and the Firefox project coding standards.

[75] Stallman, Richard, et al. GNU Coding Standards. Free Software Foundation, 8 February 2006; from: http://www.gnu.org/prep/standards/, 2006/02/16, p. 31.

[76] "cpu.c" Linux kernel, version 2.6.15.

[77] COBOL source code, COBOL-4. May 1994.

[78] "GNU Classpath Hacker's Guide." unsigned. Free Software Foundation, 5 January 2006; from: http://www.gnu.org/software/classpath/docs/hacking.html, 2006/03/01.

[79] Deimel, Lionel E., Jr. "The Uses of Program Reading." SIGCSE Bulletin, Vol. 17, No. 2, June 1985, p. 5

[80] COBOL source code, COBOL-5. October 1995.

[81] COBOL source code, COBOL-7. July 1996. Lines 01983-02032.

[82] A developer made a comment earlier in the program about the appropriateness of such detailed comments within the program, saying "!!! PLEASE REFER TO THE LEC DISCO DOCUMENT FOR MORE DETAILS ON BUSINESS RULES !!!" (COBOL source code, COBOL-7. July 1996.).

[83] COBOL source code, COBOL-7. July 1996. Lines 02112-02128.

[84] Deimel, Lionel E., Jr. "The Uses of Program Reading." *SIGCSE Bulletin*, Vol. 17, No. 2, June 1985, p. 8.

[85] COBOL source code, COBOL-6. October 1996.
I assume that there were no changes to this module because there are no signed inline comments and no entries in the change log.

[86] Program COBOL-6 was written by the same developer who was the first person to update COBOL-5 and made the remarks about "guessing how COBOL works."

[87] "Documentation Definition." unsigned. Linux Information Project, 23 February 2006; from: http://www.bellevuelinux.org/documentation.html, 2006/03/01.

[88] Gregono, P. "Comments, Assertions, and Pragmas." *ACM SIGPLAN Notices*, March 1989, p. 79-84; as quoted in: Arab, Mouloud. "Enhancing Program Comprehension: Formatting and Documenting." *ACM SIGPLAN Notices*, Vol. 27, No. 2, February 1992, p. 42.

[89] Arab, Mouloud. "Enhancing Program Comprehension: Formatting and Documenting." *ACM SIGPLAN Notices*, Vol. 27, No. 2, February 1992, p. 42.

[90] COBOL source code, COBOL-7. July 1996. Lines 01892-02032.

[91] Kernighan, Brian W. & P.J. Plauger. *The Elements of Programming Style*, 2nd Ed. McGraw-Hill, 1978, p. 141.

[92] Kernighan and Plauger, 1974, in Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Press, May 1978, , p. 199.

[93] Kernighan and Plauger, 1974, in Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Press, May 1978, , p. 199.

[94] Kernighan, Brian W. & P.J. Plauger, p. 141.

[95] Kernighan, Brian W. & P.J. Plauger, p. 150.

[96] Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." Proceedings of the 3rd International Conference on Software Engineering. IEEE Press, May 1978, p. 200.
This assertion seems to conflict directly with another, much more well-cited study. Weinberg cites an IBM sponsored "experiment in which several versions of the same code are produced, one with correct comments, one with one or two incorrect comments and one with perhaps no comments at all" (Okimoto, 1970, in Weinberg, p. 164), concluding that "for certain types of code, at least, correct interpretation of what the program does can be obtained more reliably and faster without any comments at all" (Weinberg, p. 164). This same 1970 IBM internal study continues to be cited into the second half of the 1990s, with authors using it as a basis to conclude, similar to Weinberg, that "studies of short programs show that comments in code interfere with the process of understanding, [and] if not up to date, can be misleading and cause errors in the semantic representation of the code" (Rugaber).

[97] "Commentator, The." unsigned. Cenqua Pty Ltd, 2005; from: http://www.cenqua.com/commentator/, 2006/02/15.

[98] "Documentation Definition." unsigned. Linux Information Project, 23 February 2006; from: http://www.bellevuelinux.org/documentation.html, 2006/03/01.
"[…] the fact that programmers usually do not like to write documentation and often are not good at it"

[99] "Commentator, The." unsigned. Cenqua Pty Ltd, 2005; from: http://www.cenqua.com/commentator/, 2006/02/15.

[100] "Commentator, The." unsigned. Cenqua Pty Ltd, 2005; from: http://www.cenqua.com/commentator/, 2006/02/15.

[101] "Documentation Definition." unsigned. Linux Information Project, 23 February 2006; from: http://www.bellevuelinux.org/documentation.html, 2006/03/01.

[102] "sys.c" Linux kernel, version 2.6.15.

[103] "time.c" Linux kernel, version 2.6.15.

[104] "acct.c" Linux kernel 2.6.15.

[105] COBOL source code, COBOL-1. September 1982.

[106] COBOL source code, COBOL-1. September 1982.

[107] COBOL source code, COBOL-4. May 1994.

[108] COBOL source code, COBOL-4. May 1994.

[109] COBOL source code, COBOL-7. July 1996.

[110] This concept of cyborg community derives both from the work of Donna Haraway (see: Haraway, Donna Jeane. *Simians, Cyborgs, and Women: The Reinvention of Nature*. Routledge, New York, 1991) and Gary Downey (see: Downey, Gary, Joseph Dumit, & Sarah Williams. "Cyborg Anthropology," in *Cultural Anthropology* 10(2) (1995): p. 264-269).

[111] I searched for "we" with a trailing space, making it a separate word, rather than part of another word. I spot-checked the entire 738 line return to make sure that it was consistently applied.

The total list of the files containing "we " (with the trailing space) is: acct.c, audit.c, auditsc.c, capability.c, compat.c, configs.c, cpu.c, cpuset.c, crash_dump.c, dma.c, exec_domain.c, exit.c, fork.c, futex.c, itimer.c,

kallsyms.c, kexec.c, kmod.c, kprobes.c, kthread.c, module.c, panic.c, params.c, pid.c, posix-cpu-timers.c, posix-timers.c, printk.c, ptrace.c, rcupdate.c, sched.c, signal.c, softirq.c, spinlock.c, stop_machine.c, sys.c, sysctl.c, time.c, timer.c, uid16.c, user.c, wait.c, workqueue.c

In Linux, a comment starts with "/*", but can then cross multiple lines until the end character is found by the compiler. Another search, using just the asterisk ("*") with a trailing space, provided 5743 lines of comments in the 52 files, with only 32 of those being lines, where the asterisk is being used as a multiplication sign. Further, comments in C often exist on the same line with working code, so a single line may be both a comment and an operation.

[112] "softirq.c" Linux kernel, version 2.6.15.

[113] Mulkay, Michael J. "Norms and ideology in science," in *Sociology of Science* 15 (4/5), pp. 637-656, 1976.

[114] "posix-timers.c" Linux kernel, version 2.6.15.

[115] COBOL-3 and COBOL-6, both were written by larger-than-life personalities. COBOL-6 was never updated by any other developer and COBOL-3 was updated almost exclusively by the author, with the exception of Y2K remediation. (see line 01260)

[116] COBOL source code, COBOL-3. August 1994.

[117] COBOL source code, COBOL-7. July 1996.

[118] COBOL source code, COBOL-1. September 1982.
Lower case items in the source code have been changed.

[119] COBOL source code, COBOL-7. July 1996.
Lower case items in the source code have been changed.

[120] COBOL source code, COBOL-7. July 1996.

[121] COBOL source code, COBOL-2. February 1993.
Lower case items in the source code have been changed.

[122] "Knowledge domains" comes from Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension." *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Press, May 1978.

[123] COBOL source code, COBOL-4. May 1994.

[124] Gieryn, Thomas. "Boundary-Work and the Demarcation of Science from Non-Science: Strains and Interests in Professional Ideologies of Scientists" in *American Sociological Review* 48, pp. 781-795, 1983, p. 792-93.

[125] "sys.c" Linux kernel, version 2.6.15.

[126] "sys.c" Linux kernel, version 2.6.15.

[127] Raymond, Eric Steven. "The Cathedral and the Bazaar". Version 3.0. Thyrsus Enterprises, 2000. Available online at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, accessed 2006/10/17-16:15et. Raymond writes, "My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population."

[128] "posix-timers.c" Linux kernel, version 2.6.15.

[129] "luser", *The Jargon File*, version 4.4.7. Unsigned. Available online at http://catb.org/jargon/html/L/luser.html, accessed 2006/11/18-14:51et.

[130] Turkle, Sherry. *The Second Self: Computers and the Human Spirit (Twentieth Anniversary Edition)*. MIT Press, Cambridge, Massachusetts, 2005, p. 191.

[131] "sys.c" Linux kernel, version 2.6.15.

[132] COBOL source code, COBOL-7. July 1996.

[133] see: Haraway, Donna Jeane. *Simians, Cyborgs, and Women: The Reinvention of Nature*. Routledge, New York, 1991; and Downey, Gary, Joseph Dumit, & Sarah Williams. "Cyborg Anthropology," in *Cultural Anthropology* 10(2) (1995): p. 264-269.

[134] Travers, Michael David. "Programming with Agents: New metaphors for thinking about computation." unpublished. MIT, 1996. Available at http://xenia.media.mit.edu/~mt/thesis/mt-thesis-3.4.html, accessed 2006/07/17, section 3.5.

[135] "softirq.c" Linux kernel, version 2.6.15.

[136] "configs.c" Linux kernel, version 2.6.15.

[137] "posix-timers.c" Linux kernel, version 2.6.15.

[138] Turkle, p. 247.

[139] "sched.c" Linux kernel, version 2.6.15.

[140] Turkle, p. 20.

[141] COBOL source code, COBOL-1. September 1982.

[142] COBOL source code, COBOL-1. September 1982.
Other examples at 05520-05522 and 06073-06075. All three were associated with a change log entry for PRT# 2102.

[143] COBOL source code, COBOL-2. February 1993.

[144] These graphical elements do not point to a particular word in line 00405, as I first thought. Every single instance of a comment by RTR contains the exact same elements at exactly the same spacing, regardless of the content in between the begin and end tags.

[145] COBOL source code, COBOL-4. May 1994.

[146] The developer is known to me personally and I worked with him on several projects, probably including this one.

[147] Deimel, Lionel E., Jr. "The Uses of Program Reading." *SIGCSE Bulletin*, Vol. 17, No. 2, June 1985, p. 7.

[148] Specifically, 28.34% (or 601 lines out of 2121), which was the highest percentage of any program in the sample.

[149] COBOL source code, COBOL-5. October 1995.

[150] The idea of his/her own abilities is supported by the evidence of another program in the sample, COBOL-6 (October 1996). The author of the edit in question is also the original author of COBOL-6, which is critical to the

account creation process and the successful invoicing of each account. Notably, COBOL-6 has never been edited in the ten years since it was written.

[151] Slaton and Abbate, p. 124.

[152] Turkle, p. 290.

[153] Turkle, p. 198.

[154] Brooks, Ruven, p. 200.

Vita

stuart mawler
smawler@gmail.com

Education:

Master of Science; Science & Technology Studies; Sociology Track;
Virginia Polytechnic Institute and State University, Blacksburg, Virginia 2000-2007

Bachelor of Arts in Historic Preservation; Emphasis in Architecture;
Mary Washington College, Fredericksburg, Virginia 1987-1991

Conference Paper Presentations & Publications:

International Association for Science & Technology Studies, Baltimore, Maryland 2007
Society for the Social Studies of Science (4S), Annual Meeting, Vancouver, British Columbia 2006
Bertoti Graduate History Conference, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 2006

Academic Honors:

Phi Beta Kappa National Honor Society, 1991

Experience:

Systems Architect; CSC Systems Engineering Office; IRS Contract; Intellasource, Inc. 06/2006-present

Technical Director; Enterprise Architecture; Sprint Nextel, Inc. 12/2004-04/2006

CRM Architect; Enterprise Sales & Service; MCI, Inc. 04/2004-12/2004
Senior Systems Architect & Strategist; Mass Markets IT; MCI, Inc. 04/2001-04/2004
Architect & Strategist; Mass Markets eCommerce; MCI, Inc. 11/1999-04/2001
Process Architect; Mass Markets Front Office; MCI, Inc. 04/1999-11/1999
Architect & Strategist; Mass Markets Customer Database & Order Processing; MCI, Inc. 04/1996-04/1999
Project Manager & Analyst; Mass Markets Customer Database & Fulfillment; MCI, Inc. 07/1993-04/1996
Project Manager; Commercial Calling Card Fulfillment; MCI, Inc. 08/1991-07/1993