Open Source Community in Code Comments

Abstract

This paper 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 a specific open-source sub-community. 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 completely from within the source code of the Linux kernel.

Introduction

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, serving specific purposes (for example, detailing 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 read and updated, a manuscript that is written "by hand" and pored over by other contributors and commentators, or a corpus representing shared knowledge of the group.

By design, all executable texts exist in the background of the software experience, with the intention that most people be unaware their existence. A corner ATM or word processing software are examples of software relying on many files to accomplish what looks like one task —this is software as we 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. This communication function of executable texts creates a tension between the literal and metaphoric status of programming languages.

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). Further, 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, 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. As a result much critical information is conveyed in the comments.

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 a specific open-source sub-community. These social roles are highlighted through the form and function of comments found in the texts, which emphasize the metaphoric status of programming languages and the social role of the comments themselves. Specifically, the social role of the comments is highlighted through their rhetorical construction, using a literary device that relies on the plural pronoun "we" as the subject.

Within the community and the technical literature, executable texts are not simply artifacts to be interpreted into machine-readable files and discarded. The executable texts 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 coexisting 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 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 Donna Haraway brings 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 (1986: 45-53). To a much greater degree than for Latour and Woolgar's scientists, the work of programmers within an IT department focuses "either directly or indirectly on documents" (1986: 53), 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 that is used to construct individual and group identity, rather than solely an end-product that is used to run a computer. 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 (see Illustration 1).

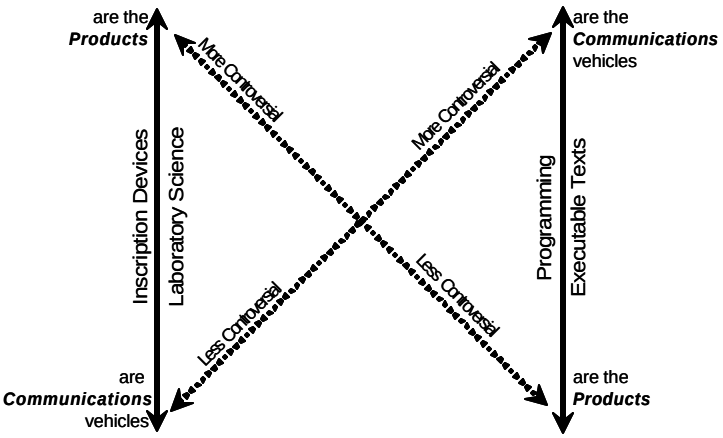

Illustration 1: Graphical depiction of the relationship of Laboratory Life *to the structure of this paper*

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* (1996: 38). 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 function of Foucauldian support.

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" (Edwards: 38). 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" (1977: 1). 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" (Barnes: 24). For programmers, an expected outcome needs to be predicted and controlled. However, even though when a system returns the appropriate outcome, the means of its creation may not be clear, even to expert programmers charged with maintenance of the system. Essentially, programming becomes non-deterministic 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" (1976: 645), 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 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. In this case, 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 (1983: 792-93). Programmers are defending their turf through comments. Their boundary struggles are many—with other sub-groups within open-source (for example, GNU/Free Software versus Open-Source or Linux versus POSIX), 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. Donna Haraway directly acknowledges this struggle, 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" (1991: 224). The form of the comments found in this study shows that this struggle extends to computer programming, where the postmodern authors associate directly with the machine, creating bodies and identities that are constructed not merely through biomedical discourse, as Haraway suggests (1991: 203), 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/C++, Java, or COBOL), the purpose of the software being built (for example, an operating system, a standalone application on a PC, or a large corporate application), and the business context in which the application is being written (for example, open-source versus proprietary). The sample studied here (the Linux kernel) represents 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 (1987: 183-84). However, similar to discourses, there are multiple competing frames operating at any given time. There is the frame of programming in the C language; the frame of programming operating systems (a more technical task); and the frame of open-source programming.

The Source Archive

To complete this study, I have relied on the source code for the Linux kernel. 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 just as tightly controlled as in proprietary software development.[1] 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 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 themes of this paper. Each comment is considered to have both form and function, where each is a continuum from normative to identity-oriented, with the latter more directly serving the programming community and discourse. Within this framework, the examples highlight many social factors, including the two overarching themes of 1) the metaphoric status of programming languages and 2) the social nature of comments and their role in reinforcing community. As a key support for the programming community and discourse, the comments show several identity-oriented literary constructions, where the language is either explicitly or implicitly structured around the inclusive pronoun "we," with various implications.

[1]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 time frame (see: http://www.kernel.org/pub/linux/kernel/, accessed 2/28/2006).

Constructing C Comments

Comments are intended to explain the symbolic code, where there is ambiguity. Richard Stallman, of the GNU free software project, requests that authors "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" (Stallman et al.: 31). This is 'good' documentation.

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" (Kernighan & Plauger: 141). However, the same author goes on to devote 13 pages "to style in commenting" (Kernighan & Plauger: 141), further stating, "[…] An excellent program […] is thoroughly commented and neatly formatted" (Kernighan & Plauger: 150). 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" (Brooks: 200).[2] It is this latter standpoint that is dominant in the programming industry, making commenting an assumed part of the normative programming practice.

While there are normative and identity-oriented forms and functions with every comment, there are considerations driven by a combination of the language used (C), the

[2]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).

environment (PC-based), and the culture (open-source) 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.

By social convention, 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" (Stallman: 31). In most programming languages, these opening short descriptions are formatted as comments and C is no exception, 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" (Brooks, 197).[3]

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.

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() & ]
//============================================================//
.(Mehta)
```

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—

[3]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.

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.

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" (Mehta). The environment in which the Linux kernel is created and maintained stresses the definitions of both "big project" and "team." In this 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.

In this 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; it is of ideological 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. Out of the set of programs, I found no instances of overtly antagonistic language, while such language was common in other source code samples I have reviewed.

Another culturally specific practice is the use of comments to mark the start and/or end of a change to the symbolic code. 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 other archives, I have found this practice quite common, but it does not exist at all within the Linux kernel.

Finally, 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" (Green).

What the commented code also reflects is uncertainty about the operation of the program and uncertainty about direction—in six months, the business direction might require the code you just removed, so don't waste it. However, if you extensively document what business reason drove the code elimination, 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.

Programming Languages

Since I have explained the structure of the comments, I will return to the programming languages themselves. As other authors have implied regarding natural language, a "programming language" may be literal or it may be metaphoric, depending on one's point of view (Semino et al.: 1272). 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 languages begins to make them look metaphoric. When looking at a specific language, programmers adopt the terms of natural language, speaking of "sentences," "paragraph," and "verbs." Even more notably, some languages use terms like "copybook," referring to entities outside of the program that are

essential to the system.[4] 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) (Gurtov: 11). 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 (Federal Standard 1037C). 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'" (Heiss), making the case that a programming language is capable of expressing all the needs of programmers. However, metaphorical borrowings like

[4]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.

jiffy and jitter seem to say that, contrary to the conceptual universe idea, "a program cannot speak for itself" (Kernighan & Plauger: 151),[5] 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) (Pflueger: 125, 136). 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*" (Scott: 5; emphasis in original). 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

[5]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'.

in symbolic logic.[6] 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. */ (acct.c)
```

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 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 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.

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 (Fauconnier & Turner: 133-87), bringing together natural language (usually English), technical

[6]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 even stronger in other archives of executable texts.

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"(Pflueger: 162) 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.

There are other ways that languages are used as metaphors that help establish community. Each new language "entails different styles of programming and suggests different modes for conceptualization" (Pflueger: 136), and may also suggest entirely different ways of forming communities around different metaphors. The name COBOL stands for "Common Ordinary Business-Oriented Language" and the language is thought of as "common" and "business-oriented." 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. 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

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 Illustration 2).

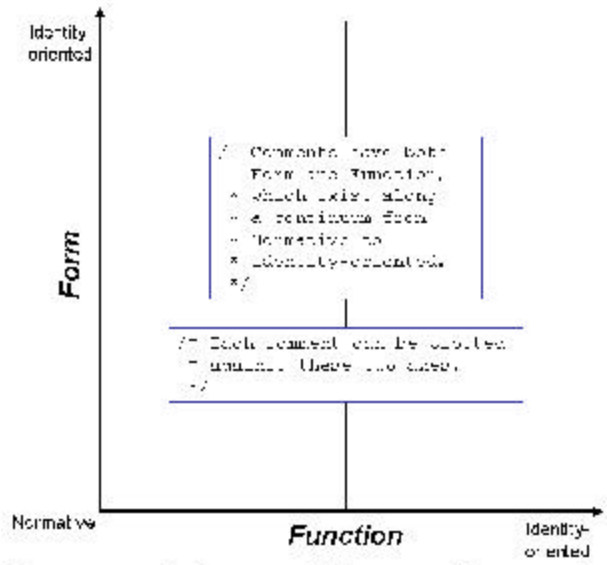

Illustration 2: Form and Function Framework for code comments

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.

The comment discussed above, showing the supremacy of natural language over the programming language, is a normative comment, regardless of its appropriateness, since it factually relates the information about the size of the mantissa, the base used for the exponents, and the use of the relationship of the mantissa to create the maximum fractional value. That comment is best plotted in the lower left (at the origin), representing the most fully normative form and function.

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!” (futex.c)
```

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.

Letting the Executable Texts Speak

Despite the humor content of the futex example above, a conversation seems like an even clearer way to show that the medium is not a static history of changes to the program; 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
 */(sys.c)
```

The above comment 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, 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. 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, potentially placing this example in the upper right quadrant 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.
 */(time.)
```

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);
}(acct.c)
```

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.

“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, the literary forms allow elevation of the programmer (and his/her discipline) from technician to teacher, associating programming discourse with academic discourse. Second, the literary forms help maintain boundaries with outside individuals, organizations, and specifically users. Finally, the literary forms expand the group identity to include not just the programmers, but the machines and software systems they create and manage as a cyborg community (Haraway, 1991; Downey, et al., 1995).

Group Elevation

In some ways, group elevation is the most subtle, but the most pervasive. The comments are highly normative in function and yet identity-oriented in form, though falling within the lower left quadrant 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.
 */(softirq.c)
```

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” (1976: 637-56), where the grammar justifies elevation of programmer status to that of professor or leader.

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. 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 referred to as the "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];(sys.c)
```

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;(sys.c)
```

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. Open-source programmers, in particular, tend to be among the elite, so their greater technical understanding may have merit. That this language would imply wizard status for programmers is coherent with the elite group or secret society, since they possess rarefied knowledge.

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.
 */(sys.c)
```

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.

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 (Haraway; Downey, Dumit, & Williams). 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" (Travers).

```
/* Some compilers disobey section attribute on statics when not
   initialized -- RR */(softirq.c)
```

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 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          */(config.c)
```

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 in 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.

The association of programmer with the machine is completed 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.
 */(sched.c)
```

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.

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.

Conclusion

Comments have an "official" or normative purpose—explain how the program works—but close reading of the programs in which they occur 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, and stretch the definition of programming community to become a cyborg community, including the systems the programmers create and the machines on which they run.

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" (Brooks: 200). Instead, recognizing the dual normative/technical and social/identity-oriented form and function of comments is a more illuminating way to approach both a social group and a form of communication.

Bibliography

Source Archive

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.

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.

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.

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

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.

“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.