The discursive coding of software: a study of the relationship between stability and change

Jennifer Helene Maher
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Discourse and Text Linguistics Commons, Other Communication Commons, Programming Languages and Compilers Commons, and the Rhetoric and Composition Commons

Recommended Citation

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
The discursive coding of software:
A study of the relationship between stability and change

by

Jennifer Helene Maher

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Rhetoric and Professional Communication

Program of Study Committee:
David Russell, Major Professor
Carl Herndl
Dorothy Winsor
Diane Price Herndl
David Schweingruber

Iowa State University
Ames, Iowa
2006

Copyright © Jennifer Helene Maher, 2006. All rights reserved
# TABLE OF CONTENTS

Abstract................................................................................................................................... iv

Chapter One: The Discursive Coding of Software .............................................................. 1
  A Short History of Software .................................................................................................. 5
  The Symbolic power of the Software Programming culture ........................................... 11
  Needs My Research Addresses ....................................................................................... 15
  Research Questions ......................................................................................................... 18
  The Relationship between Discourse and Culture ......................................................... 19
    Discourse ........................................................................................................................ 19
    Culture ............................................................................................................................. 24
  Overview of Chapters ....................................................................................................... 28

Chapter Two: Theory, Framework, and Methods ............................................................ 31
  Theories of Practice .......................................................................................................... 32
    Structuration Theory ..................................................................................................... 33
    Articulation Theory ....................................................................................................... 35
    Constructivist Structuralism ......................................................................................... 37
  Critiques of Theories of Practice ...................................................................................... 39
  The Spatio-Temporal Framework ................................................................................... 44
    Time as Temporal Orientations to the Past, Future, and Present .................................. 46
    Space as Social Positionings ......................................................................................... 52
  Methods and Choice of Texts ............................................................................................ 57
    Iteration and Synthesis of Software Literature: Chapter Three .................................. 57
    Projectivity and Ideological Analysis: Chapters Four and Five ..................................... 59
    Practical Evaluation and Genre Analysis: Chapter Six ............................................... 66

Chapter Three: The Reactivation of Iterative Social Practices ........................................ 69
  The Practice of Programming as Knowledge Work ....................................................... 70
  Instrumentalist Talk about Coding Issues ....................................................................... 77
    Quality ............................................................................................................................. 78
    Security .......................................................................................................................... 79
    Flexibility ....................................................................................................................... 79
    Innovation ...................................................................................................................... 80
    Transparency ................................................................................................................ 81
  The Discursive Practice of Evangelism .......................................................................... 81
  Regulatory Practices ........................................................................................................ 82
    Trade Secrets ................................................................................................................ 83
    Patents ............................................................................................................................ 84
    Copyright ....................................................................................................................... 85
The Creation of Functional Differences in Software .......................................................... 88
Established Software Technology ................................................................................... 89
Disruptive Software Technology .................................................................................... 92

Chapter Four: An Ideological Analysis Of the Encoding of Software ......................... 95
Stories of Software .............................................................................................................. 96
  The Entrepreneurial Narrative of Gates and the Professionalization of PC Hobbyists .. 98
  The Idealistic Narrative of Stallman and the Free Software Movement ...................... 111
  The Pragmatic Narrative ............................................................................................... 123
The Evolution of the Competing Ideological Trajectories ............................................. 134
  Software as a Commodity: Gates’ Evangelism ............................................................ 135
  Software as Morality: Stallman’s Evangelism .............................................................. 139
  Software as a Tool: Torvalds’ Evangelism ................................................................... 143

Chapter Five: Imagining Software Through Identity in Competing Communities of the
Software Programming Culture ...................................................................................... 148
Identity in the Network Society ........................................................................................ 150
Identity in the Software Programming culture .............................................................. 153
  The Legitimizing Identity and the Proprietary Software Industry .............................. 155
  The Resistance Identity and the Free Software and Open Source Communities ......... 165
  The Free Software Community as a Resistant/would-be-Project Identity ............... 168

Chapter Six: Reconfiguring Copyright and Software Licensing Agreements as Strategic
Deliberative Action ............................................................................................................. 185
The Generic Application of Software Licensing Agreements ........................................ 187
  Copying and Distribution .............................................................................................. 192
  Modification .................................................................................................................. 206
  Warranty ....................................................................................................................... 209
Enforcing Software Licensing Compliance .................................................................... 214
  Enforcing the EULA .................................................................................................... 216
  Compliance and the GNU GPL .................................................................................. 218
  Compliance and the OSD .......................................................................................... 219

Chapter Seven: Toward a Digital Literacy ...................................................................... 221
References ....................................................................................................................... 229
ABSTRACT

Software has become an integral part of most people’s everyday lives. Though discussions of software in the field of rhetoric have traditionally taken an instrumentalist approach that seeks to increase the effective use of software, a critical turn has established the need to examine the ideological signification of software. I argue that one way we can do so is to examine the social practices that symbolically constitute software to reveal how software is discursively coded, and how, consequently, users of software are themselves discursively coded. To this end, I examine the discourse of those primarily responsible for the design and regulation of software—the software programming culture.

To analyze the discourse of the software programming culture, I developed a spatio-temporal framework rooted in theories of practice and theories of space and time to guide both my choice of data and the methods by which I conduct analyses. In attempting to resist the objectivist/subjectivist dualism that so often limits accounts of how meaning is made through social practices, I employ historical, ideological, and genre analysis to contextualize and analyze the discourse of the software programming culture. My goal is to identify, describe, and map a network of meaning that simultaneously allows for stability and change. I argue that this network of meaning constitutes not just software but also the software programming culture as a discursive site of ideological struggle, the effects of which have significance for society as a whole.

Unveiling the process by which software, as a technological and culture artifact, is discursively coded is a move toward digital literacy that is important for understanding the ways in which ideologies are embedded in the discourse of technological artifacts. In the case of software, I argue that such an understanding may allow even the least technologically-adept users to interrupt and challenge their own coding as digital subjects by the software programming culture and to cultivate a digital literacy that empowers users to have a voice in how their own lives are affected by the discursive coding of software.
CHAPTER ONE: THE DISCURSIVE CODING OF SOFTWARE

...there is a group of software developers in the United States, and other parts of the world, that do not believe in the approach to copyright protection mandated by Congress. In the past 20 years, the Free Software Foundation (FSF) and others in the Open Source software movement have set out to actively and intentionally undermine the U.S. and European systems of copyrights and patents. Leaders of the FSF have spent great efforts, written numerous articles and sometimes enforced the provisions of the GPL [the General Public License] as part of a deeply held belief in the need to undermine and eliminate software patent and copyright laws.

Darl McBride (December 4, 2003), owner of the proprietary software company SCO which holds the copyright for UNIX, commenting on his company’s lawsuit against IBM, the biggest corporate supporter of the GNU/Linux operating system

Maybe someone can explain to Darl that the GPL is designed so that people receive the value of other people’s copyright works in return for having made their own contributions. That is the fundamental idea of the whole license—everything else is just legal fluff...So not only is Darl wrong when he attacks the GPL as being somehow against “financial gain;” the notion that the GPL has, of “exchange of receipt of copyrighted works,” is actually explicitly encoded in U.S. copyright law. It’s not just a crazy idea that some lefty Commie hippie dreamed up in a drug-induced stupor.

Linus Torvalds (December 8, 2003), the original developer of the Linux kernel and leader of the Open Source community, in response to McBride

But in the end, it is our ability to unify all of the elements of the information society—software, hardware, and bandwidth—in shared hands, that is our own hands, that determines whether we can succeed in carrying out the great 18th century dream, the one that is found in Article 1 Section 8 of the United States Constitution, the one that says that human beings and human society are infinitely improvable if only we take the necessary steps to set the mind free. That’s where we are really going. Mr. McBride’s company’s fate, whether it succeeds or fails...is compared to that. We are running a civil rights movement. We’re not trying to compete everybody out of business, or anybody out of business. We don’t care who succeeds or fails in the marketplace. We have our eyes on the prize. We know where we are going: Freedom. Now.

Eben Moglen (February 23, 2004), member of the FSF Board of Directors and lawyer for the FSF, in response to McBride

These epigraphs, chosen from a distributed conversation among representatives of what I term “the software programming culture,” revolve around a March 3, 2003, complaint filed by software vendor SCO, formerly known as Caldera Systems, Inc., against International Business Machines, more commonly known as IBM, for a series of violations against SCO’s copyright on the UNIX operating system. On its face, the conversation is about copyright law and software, the instructions according to which all computerized technologies perform tasks.
However, the meaning of this conversation among McBride, Torvalds, and Moglen extends far beyond the writing, or coding, of software and the practices regulating its exchange to the politics and ideologies behind code development and the circulation of software. The competing discourses evident in the exchange point to the connections between the development and dissemination of software and broader social, political, and ideological issues. These issues include ownership, nationalism, the meaning of value, and the effect of computer technologies on (re)envisioning freedom in the networked society of the Information Age. Consequently, if those of us outside the borders of the software programming culture—users and consumers of software and software-driven products—are to understand what is at stake in the struggle over the meaning-making of software that occurs through what I call the discursive coding of software, we must understand the ways in which we are inscribed both materially and ideologically by the discourse represented in such conversations.

Let me start by explaining what I mean by the discursive coding of software. When considering the coding of software, what immediately comes to mind is machine language: technical combining and recombining of ones and zeros, or the manipulation of any other system of signs that enables a computer to function. However, such a functional understanding of code does not account for the ways in which this most fundamental encoding represents only one layer of meaning that we can assign to the “coding” of software. Software acts upon reality “beyond the box” of the computer, not through machine language but through natural language—through discourse. Through their natural language discourse about software code, the developers and regulators of software—the architects of machine language—“encode” (in the sense in which semiotics uses the term) social actions that are shaped by the products, ideas, and activities that are made possible through the use of software. Whether these actions entail programmers talking about development processes, office managers discussing whether to buy the Windows or Linux operating system, or students deciding what web browser to use, each act is in large part encoded through the discourse of the architects of software and the culture to which they belong, the software programming culture. The discourses surrounding the production, use, and circulation of software contribute layers of meaning beyond the functional meaning of its sign that is
encoded through competing ideologies articulated by and within the software programming culture. Software, in short, is made to mean something in the world beyond ones and zeros.

Those at ground zero of the site of struggle over how software is made, distributed, and circulated (and how the rest of us will live with computers) are members of the software programming culture. As Grossberg (1996a) argues, “Culture is the struggle over meaning, a struggle that takes place over and within the sign...But it is not only the sign that must be made to mean, it is the world as well...Culture is the site of struggle to define how life is lived and experienced” (p. 157-58). Further, a culture provides to its members a set of typified social practices through which to act and give meaning to actions, what I refer to as “cultural practices.” Through the cultural practices available to the members of the software programming culture, these architects encode software with meaning not just for themselves but also for society. Because these practices have been sanctioned, even naturalized over time through their consistent use by actors within the software programming culture, they are temporarily stabilized resources available for inscribing meaning. The establishment of stability is, in fact, necessary for the existence of the culture and for meaning within the culture to evolve. Paradoxically, the shared understandings that are temporarily stable, or, in the words of Schryer (1999) “stabilized enough” (p. 108), lead to differences in the ways that ideologies are encoded into practice producing a plural rather than monolithic software programming culture. Thus, the architects of software have resources by which to both code software and encode the cultural practices surrounding its development and use.

In commenting on cultural production generally, Hall talks about the plurality of codes and encoding, a condition that “does not destroy the process of encoding, which always entails the imposition of an arbitrary ‘closure.’ Indeed, it actually enriches it, because we understand meaning not as a natural but as an arbitrary act—the intervention of ideology into language” (Grossberg, 1996a, p. 137). Extending our understanding of software beyond its functional, or instrumental, meaning allows us to recognize the role that agency can have in both the process of encoding and decoding. By decoding—rhetorically analyzing—the ways in which ideology is encoded in software differently by various architects of the software programming culture, we, as coding and coded subjects, open up opportunities to intervene and participate in the struggle to assign meaning to software and its effects on the
society, culture, and everyday life. Therefore, describing and analyzing the intervention of ideology into language within and by the software programming culture, what I call the discursive coding of software, is the aim of this dissertation.

Software has become an integral part of most people’s everyday lives, controlling everything from cell phones, personal computers, and medical devices to the Mars Rover, nuclear submarines, and voting machines. And though software typically functions as intended, or without major consequence, software-related problems have occurred with significant effects. For example, between June 1985 and January 1987, the software configuration for the Therac-25 medical accelerator, a radiation therapy device, caused the lethal overdose of at least five patients (Leveson & Turner p.18). On February 25, 1991, a Patriot missile defense system in Dhahran, Saudi Arabia, failed to intercept and track an incoming Scud missile during the first Gulf War because of a problem with the software in the system’s control computer, according to a U.S. Government Accounting Office report (p. 1). 28 Americans were killed when the Scud hit an Army barracks. Though the problem had been identified weeks beforehand, the modified software did not reach Dhahran until the day after the fatalities occurred. And a 2002 National Institute of Standards and Technology report, commissioned by the U.S. Department of Commerce, estimated that software errors annually cost the national economy $59.5 billion (p. ES-3).

But the effects of software, even such significant effects, extend beyond these “tools” of the networked society. As Feenberg (1999) suggests, computerized technology, which functions according to the software that instructs it, “is not merely the servant of some predefined social purpose… [but] is an environment within which a way of life is elaborated” (p. 127). The architects, more so than the inhabitants, of this environment of technology, which now elaborates a way of life that involves most of us in one way or another, have created new ways of working, communicating, and envisioning the world. They have developed a vocabulary of hardware and software, a logic of pathways and circuits, a language of code. In essence, they have evolved a globalized social system of computer technologies as well as, in large measure, much of the discourse that defines and describes it.

To provide essential background for my analysis of the discursive coding of software by the software programming culture, I first present a short history of software. This history
situates the emergence of the software programming culture that shaped the competing discourses that developed within it as well as the practices that inform the discursive coding of software.

A SHORT HISTORY OF SOFTWARE

Today, software is an integral component of the networked society’s infrastructure, enabling many of the day-to-day activities of life. The invisibility, pervasiveness, and ubiquity of software that we now take for granted would have been hard for anyone other than electrical engineers and computer scientists to imagine in the early days of room-sized computers. In the nascence of computers in the 1940s, computer instructions consisted of series of holes punched onto cards or paper tapes. Individuals fed these punch cards into calculator computing machines that performed advanced mathematical equations as instructed by the cards. The usefulness of computers was limited to addressing the needs of the few whose work involved complex calculations.

The 1950s marked the development of programming tools and languages that allowed programmers to write code in ways more like natural language, which could then be translated into the ones and zeros of machine language. Tools such as interpreters, assemblers, and compliers allowed for this translation in the next-generation mainframe computers that were used not only for mathematical equations but also for governmental, academic, and commercial work. In addition, the artificial languages of programming, which have semantic and syntactic characteristics of natural language, allowed programmers to write computer instructions more easily and quickly. In 1957, IBM released FORTRAN, a high-level programming language for its 704 mainframe computer. ALGOL, remarkable in that it was the first programming language not specific to a particular piece of hardware, soon followed FORTRAN. Also in 1957, the first time-share operating system was developed. This OS allowed users to share the same mainframe, which only the wealthiest of companies, universities, governments, and military organizations could afford. About the advancements of the 1950s, computer historian Ceruzzi (1998) writes, “[computer pioneers] and their customers slowly realized: first, that [programming] existed; second that it was important; and third, that is was worth the effort to builds tools to help do it. These tools combined with
the application programs, became collectively known as ‘software,’ a term that first came into use around 1959” (p. 108).

With the development of the minicomputer in the 1960s, computers reached a completely new audience. Though mainframes and the software punch cards that instructed them remained the standard for those who had large processing and database needs and sufficient capital, minicomputers introduced the idea of the computer as a “personal interactive device” (Ceruzzi, p. 124). Significant software contributions of the 1960s that continue to be in use today include programs that enables a mouse to function, the graphical user interface, word processing, and even spell-check. Software computing became a formal academic major in this decade, and in 1968 at a computing conference in Germany, the term “software engineering” was first used. A year later, IBM, which dominated the computer market, decided to “unbundle” its software from its hardware, requiring software to be purchased separately. As a result of this action—motivated in part by an antitrust lawsuit filed against IBM by the U.S. government because of IBM’s dominance in the mainframe computer market—software came to be seen as a commodity in its own right rather than as a free package that came with the hardware (Ceruzzi, Campbell-Kelly). Thus IBM, whose revenue continued to derive primarily from mainframe and minicomputer sales, became the largest and most profitable supplier of software (Ceruzzi), though by no means the only supplier.

The 1970s ushered in the possibility of personal computing for the masses, and it is in this decade that I begin my analysis of the discursive coding of software by the software programming culture. In addition to the development of distributed network email via ARPANET (the precursor of the Internet), software portable to and compatible with different hardware types, spreadsheet applications, and distributed newsgroups (USENET), hardware developments opened up the possibility of software use on a new scale. In January 1975, Micro Instrumentation and Telemetry (MITS) began selling the Altair 8800, the first microcomputer, or personal computer, as a $400 kit to computer hobbyists.

I locate the birth of the modern software programming culture with the development of the Altair 8800. The Altair 8800 didn’t look or function anything like the personal computers, or PCs, of today; it was, in fact, nothing more than a nondescript box that, when
programmed through the flipping of switches on the front of the box, created a sequence of flashing lights. But the Altair 8800 had an Intel microprocessor chip, the precursor to those most computers have today. The 8080 microprocessor enabled the Altair to function as a general-purpose computer with storage capabilities that, though incredibly limited, allowed for memory. (A floppy disk drive could be ordered with the Altair, as well as other PCs such as the Apple II and Radio Shack TRS-80 in 1977. The Apple II and TRS-80 included a keyboard and monitor, making both more appealing to users.) However, because no software came with the Altair, a small start-up company founded in 1975 called “Micro-Soft” (taken from the words “microcomputer” and “software”) seized the opportunity to revamp the programming language BASIC for the Altair. Thus Microsoft, along with other software development companies, began the proprietary software industry for the PC.

Hobbyists, referred to then as “hackers” but without the criminal associations common today, also became an important influence on the evolution of both computers and software. The Homebrew Computer Club first met on March 5, 1976, in a member’s garage in Menlo Park, CA, and additional Homebrew Computer Clubs began to form around the country soon after. This “amateur computer users group,” as an announcement in the club’s first newsletter described the organization, was created as a forum for people with “likeminded interests” in computers, the “digital black-magic box” (Moore, 1976). The availability of the Altair 8800 served as the impetus for the first Club, formed to support enthusiasts indulging their hobby. The Club’s fourth newsletter recounted a meeting at Stanford University, explaining, “‘It’s a hobby.’ Yes, a hobby for fun. Interest in home computing is spreading fast…By sharing our experience and sharing tips we advance the state of the art and make low cost home computing possible for more folks” (Moore, 1975). Describing the later part of this decade, Ceruzzi writes, “There was a strong and healthy industry of publications, software companies, and support groups to bring the novice on board. The personal computer had arrived” (p. 240). With the availability of the PC, which made individual ownership of computers possible, hobbyists also stood poised to become “software suppliers”.

The 1980s is the start of software as we commonly know it today. PCs became a worldwide business phenomenon and eventually a social phenomenon. For software, the
early 1980s was a time of intense development, what computer historian Campbell-Kelly (2003) identifies as “gold rush” years. Microsoft continued for PCs and Apple the software development it had begun for the earliest microcomputers in the late 1970s. Most notably, in 1980, Microsoft licensed an operating system MS-DOS based on software written by Tim Paterson of Seattle Computer; MS-DOS turned out to be “one of the longest-lived and most-influential pieces of software ever written” (Ceruzzi, p. 270). The popular video games *Space Invaders* and *Pac Man* were released at the beginning of this decade for Atari. And in December 1982, *Time Magazine*’s Person of the Year was replaced by the “Machine of the Year,” IBM’s Personal Computer (PC), the name that would become as synonymous with microcomputers as Kleenex is with tissues. In 1983, Microsoft released the word-processing program *Word*, and in 1985, the long-awaited Microsoft Windows 1.0 was released. Also during this period, UNIX, originally developed in 1969 by AT&T as an internal software program, became the most popular OS in the world because of the use of UNIX variants on office workstations and in educational institutions.

In response to the proliferation of proprietary software, Richard Stallman began in 1983 the GNU Project with the goals of providing free software and “creating a new software-sharing community” (2001). In 1985, Stallman founded the Free Software Foundation (FSF), an organization dedicated to these goals. In 1989, Michael Teimann founded Cygnus Solutions, the first free-software distribution and service company. These acts and others like them signaled a new counter-movement in PC software—the Free Software Movement, which I discuss in detail in subsequent chapters of this dissertation.

Also during the 1980s, software that made the Internet what it is today proliferated. In 1984, the free software Domain Name System (DNS) that translates the natural language web addresses (e.g. www.amazon.com) into the numerical addresses that hardware reads was developed. And in 1989, Tim Berners-Lee made hypertext distributable via the Internet possible by developing the http protocol and addressing schema URLs, which, like the DNS, are important free software contributions to the Internet and its proliferation as an everyday tool.

Software developments throughout the 1990s contributed significantly to software proliferation and use. In 1991, Linus Torvalds began work on a UNIX-compatible kernel (the
core of an operating system that allows hardware and software to communicate) that he called “Linux”. When this kernel was used in conjunction with software from the GNU Project in the following year, the “GNU/Linux” or “Linux” operating system was born. (In this dissertation, I refer to the operating system as “GNU/Linux,” unless quoting from or discussing a text, actor or organization that uses “Linux” to represent the whole operating system. In Chapters Four and Five, I discuss the significance of the issue of naming this operating system.) Red Hat, the most successful Linux distribution company to date, was subsequently founded in 1994. In the same year, the web browsers Netscape Navigator, followed by Microsoft Internet Explorer in 1995, allowed for easy-to-use graphical interfaces for “riding” the Internet. In 1996, the free K Desktop Environment, more commonly known as KDE, offered users the ability to customize the look of their desktop or mimic the graphical interface provided by Microsoft Windows, which had become in this decade the most popular operating system in the world.

The 1990s also mark a particularly contentious period in the history of software. The PC and the software that commands it, as well as other computer technologies, had become an integral part of the infrastructure of everyday life. Because of its pervasiveness, a struggle over and through software began. At issue was not only the functionality of software, but also ideological issues that have long occupied society including freedom, consumerism, nationalism, to name only a few.

By 1998, Microsoft had succeeded IBM as the number one supplier of proprietary software in the world. However, this success was not without consequence. In the same year, the U.S. Department of Justice, in conjunction with what would eventually entail 20 states, charged Microsoft with violating the Sherman Trust Act (United States of America vs. Microsoft Corporation). In short, Microsoft was accused of being a monopoly and engaging in practices that strengthened that monopoly at the expense of consumers and competitors (e.g., the bundling of Microsoft Internet Explorer with the Windows operating system). While this most public of court cases played out, Stallman, one of the earliest critics of the kind of software development and regulation that had allowed Microsoft to become a monopoly, continued his cry for software to be “free,” a process that would both prevent the circumstances that allowed Microsoft to become an alleged monopoly and would further
moral idealism. But, while Stallman advocated free software, many who also developed free software became put off by Stallman’s ideological construction of free software. In 1997, to differentiate Stallman’s ideology from the practical benefits of free software, a group of programmers began using and advocating the word “open” to describe the practical, and thus, according to this group, the non-ideological processes that made free software free software. This naming signaled the birth of a new counter-movement in the software programming culture—the Open Source Movement. At the beginning of 1998, the Open Source Movement won its first public victory when Netscape released the source code for Netscape Communicator, what had been the most-often used web browser before Microsoft began its alleged monopolistic practice of bundling Internet Explorer with Windows. At the close of the decade, Microsoft was found guilty of violating the Sherman Trust Act—the world’s leading supplier of software was a monopoly.

In 2000, U.S. vs. Microsoft resulted in the order to break up Microsoft, which in 2001 was whittled down to an agreement that allowed Microsoft to continue bundling any of its software with the Windows operating system. But in a move that many saw as a victory for open-source, in the same year Microsoft released its Shared Source Philosophy. This approach to software, according to Microsoft, entailed “a balanced approach that allows us to share source code with customers and partners while maintaining the intellectual property rights needed to support a strong software business” (Microsoft, 2002b). Though many in the Free and Open-Source movements did not see the Microsoft Shared Source Philosophy as sharing enough, Microsoft’s new “balanced approach” clearly signaled a change in the dominant construction of PC software as a proprietary commodity. Although Microsoft’s Shared Source Philosophy was clearly important, its effect at this point is still very limited.

Free and open-source software has received its own share of challenges with its gaining popularity. In 2003, the SCO group (formerly Caledera Systems Inc.), the current copyright owner of UNIX, began a lawsuit against IBM that eventually centered on the issue of copyright infringement (Caldera Systems Inc. vs. International Business Machines Corporation). SCO argues that the Linux operating system includes source code taken from UNIX. As the chief supporter of “Linux,” IBM is charged with violating SCO’s copyright. Though this case has yet to be decided as of August 2006, its implications pose a serious
threat to free and open-source software as well as to corporations, governments, and individuals who use them.

As this short history of software illustrates, the evolution of software is easily chartered by a series of innovations in its nascence in the 1940s to the 1990s. And though innovation continues today, the pervasiveness of software clearly led to a shift in the 1990s, a shift in which software became less about functionality and about something more. As the US vs. Microsoft and Caldera Systems Inc. vs. IBM illustrate, there is much at stake, both materially and ideologically, for members of the software programming culture and, consequently, for anyone who uses licensed-software (a discussion that occupies Chapter Six). What exactly is at stake in the struggles that occur within the software programming culture is what I seek to uncover in this dissertation.

**THE SYMBOLIC POWER OF THE SOFTWARE PROGRAMMING CULTURE**

In spite of the fact that software is everywhere, its pervasiveness is still relatively invisible. As a result, the ideological signification of software through its discursive coding is cloaked in this invisibility. My analysis seeks to unveil the ideological signification and the ways in which the discursive coding of software occurs through the cultural practices of the software programming culture. Attention to the discursive coding of software is particularly important, I argue, because of the high degree of autonomy that the programming culture has enjoyed due to—and at the expense of—lay users who lack the digital expertise of those who create the technology. For most lay users, software works well if, and only if, they are not forced to give it any notice. Software is, after all, designed to be invisible, hidden away behind icons and common pathways culled from everyday life. We, as lay users, want software to be invisible because when it is not, there is typically a problem that is difficult, if not impossible, to solve on our own without technical expertise. As Winner (1978) suggests, “technology succeeds if it is made to appear autonomous” (p. 15) and software has certainly succeeded in this enterprise, if not entirely without notice. For example, Lessig (1999), in his discussion of the Y2K “environmental disaster,” (p. 54) locates the reason for this disaster with the software programming culture. Because programmers have been allowed “to think that their actions are their own,” an epistemology that is reinforced by “cultural and legal systems” that take a hands-off approach to software development, these architects of code are
not held accountable for faulty program writing such as that which led to the Y2K hysteria (Lessig 1999 p. 54-55). The insularity and autonomy of this culture, whose work provides the infrastructure of our everyday lives, results from the symbolic power—the “invisible power which can be exercised only with the complicity of those who do not want to know that they are subject to it…” (Bourdieu, 1993, p. 164) —with which we have empowered software programmers. This empowerment at our own expense is particularly dangerous because it has become increasingly difficult to live our lives without software and the tasks it allows us to complete.

Although Lessig argues that studies of technology have been around since the development of technological artifacts, software is unique in its scope—code makes possible many very different tasks of modern living, even the most mundane. This fact necessitates a re-imagining of what it means to study pervasive technologies such as software. As Lessig notes, “Code may be only a difference in degree [from other technologies], but a difference in degree at some point becomes a difference in kind” (p. 232). A reason for this difference in kind arises from one aspect of the discursive coding of software, what Lessig terms “code talk,” the tendency to talk about software and its effects in the world “as if the worlds I am describing were in some sense elsewhere” (p. 101). Because code remains invisible within computer hardware and behind the tasks it performs, it is easy to ignore its very real effects and implications in everyday life. Even when a software-related problem does occur, we do not see the faulty code itself but rather an error message that signifies but does not make visible the failure of the code. In the case of Microsoft Windows, the most commonly used operating system for PCs, an often-seen representation of code failure is the dreaded blue screen that appears on the monitor and represents an error occurrence in the software. To many lay users, the blue screen is nothing more than a glitch in their lives as digitalized subjects. The blue screen represents an error, but not necessarily one that can be known and corrected by the user. In fact, the actual error often remains a mystery, beyond the lay user’s purview.

Lessig argues that critical approaches to what he calls “code-talk” must render visible the effects of software in our lives:
We live in a real space, subject to the effects of code. We live ordinary lives, subject
to the effects of code. We live social and political lives, subject to the effects of code.
Code regulates all these aspects of our lives, more pervasively over time than any
other regulator in our life. Should we remain passive about this regulator? Should we
let it effect us without doing anything in return? (p. 233)

Investigating the practices of the culture from which the technology of software emerges and
derives meaning provides opportunities to challenge further the stance of technological
neutrality and to emphasize the political ideology of software. This emphasis creates
opportunities for critical understandings of technology that can lead to digital literacy and
deliberately meaningful action by users despite a lack of technical expertise. Digital literacy
and meaningful action occurs not just through access to and use of computer technologies
but, more importantly, participation in conversations about the discursive coding of software.
In examining the cultural practices that shape and are shaped by the technological
environment of software—or more aptly, the discourse of software exchanged within the
culture that develops software—this dissertation seeks to uncover the political ideologies
embedded in and disseminated through the discourse of the culture as moments in which the
social practices that (re)produce and circulate the ideologies come together in struggle.

Researchers in the field of rhetoric are in a unique position to critically examine the
discourse of the software programming culture because of our interest in the ways in which
social practices, particularly discursive practices, function in the making of meaning in the
world. However, much of our research into software has focused on increasing the
effectiveness of software use by examining issues of documentation (see Liebhaber, 2002;
Selber et al., 1996; Walters and Beck, 1992; Barker, 1990; Pierson et al., 1988), usability (see
Mirel, 2002; Mirel and Olsen, 1998; Mehlendbacher, 1993), or some combination of the two
(see Guillemette, 1989; Velte, 1989). Walters and Beck, for example, argue for a “rhetorical
stance” to software documentation that improves the efficacy of the documentation for users
through the inclusion of examples, structural frameworks, and persuasive writing techniques.
Mirel and Olsen, in another efficacy-oriented discussion of software, advocate coursework
for software engineering students that foregrounds writing so that software developers can
learn to adopt “user-centered beliefs and design practices” (p. 197). Adopting a user-centered
approach, which is the exception rather than the rule in software engineering, stands to help designers “infuse usability into every aspect of user and task modeling, program design, development, and testing,” (Mirel & Olsen p. 197) and thus, improve the likelihood of effective use of the software. While useful and certainly important, instrumentalist research that centers on increasing the effectiveness of software does not provide, nor does it seek to provide, a critical understanding of software, the contexts in which software is developed, and the social practices by which software is given meaning. Thus, it does little to illuminate the ideologies associated with the development and use of this particular kind of technology.

Yet, through a critical turn in rhetoric studies, researchers have increasingly noted a need to turn a critical eye toward the study of software and/or to the culture in which software is developed (see Selber, 2004; Haefner, 1999; Temple Dennett, 1998; Lay, 1996; Selfe & Selfe, 1994). For example, Selfe and Selfe, in their study of computer interfaces, argue that compositionists must realize the implications of technology in education to help students become “technology critics”: “with such a realization, we maintain, English composition can begin to exert an increasingly active influence on the cultural project of technology design” (p. 484). Lay has called for researchers to acknowledge the role of gender identity and difference inherent in the “computer software programming/developing culture,” as well as in the computer culture as a whole, and to understand the implications of these differences in the research and practice of nonacademic writing. Selber suggests that “critical literacy” can be fostered by attention to the “design cultures” that create and maintain computer infrastructures, of which software is an integral part; to the specific environments or “use contexts” in which a particular infrastructure is utilized; to larger “institutional forces” that shape these contexts; and to “popular representations” of software, such as those put forth by the media about Microsoft (p. 106-133). Unlike “functional literacy,” which emphasizes the effective use of computer technology, critical literacy is a mode of thinking whereby students become “questioners of technology” rather than passive users through uncovering and challenging the values and ideologies of the “status quo” (Selber, 2004, p. 1982). With such questioning, students stand poised to translate their critical literacy into social action, which in Selber’s study takes the form of students as “reflective producers of technology” through the rhetorical “design and evaluation of online environments” (p. 1982).
Critical turns such as Lay’s and Selber’s extend discussions of software beyond effective use to address critically the social practices and ideologies that influence ways in which computer technology is produced, identified, regulated, consumed, and circulated. It is through such a crucial understanding that digital literacy, or the critical literacy of digital technologies, can develop. Winner (1988) notes the importance of understanding technologies in a socio-historical framework that encourages a critical approach to technology, “What matters is not technology itself, but the social…system in which it is embedded… [this is called] the social determination of the technical [which] emphasizes looking behind technical devices to see the social circumstances of their development, deployment, and use” (p. 21). I argue that we can “look behind the devices” by examining the cultural practices that symbolically constitute the software programming culture to reveal the ways in which software is discursively coded by its architects. Unveiling the process of discursive coding is thus a move toward the kind of critical literacy that Selber argues is important for understanding the ways in which dominant ideologies are embedded in the discourse of technological artifacts and for challenging these ideologies. In the case of software, I argue that such an understanding may allow even the least technologically-adept lay users to cross one seemingly impenetrable boundary of the digital divide and to cultivate a digital literacy that empowers them to understand and even have a voice in how these technologies affect their lives.

**NEEDS MY RESEARCH ADDRESSES**

With the aim of fostering critical digital literacy, in this dissertation I examine the discursive coding of software and the cultural practices of those who have thus far been primarily responsible for the encoding of software—the software programming culture. This examination addresses two related needs in critical research on technology:

1. To study critically the contexts in which computer technologies are developed.
2. To develop theories, methodologies, and frameworks that encourage a complex understanding of technology in context.

Though many researchers have undertaken critical studies of computer technologies (see Salvo, 2002 on computer pedagogy; Johnson-Eilola, 1997 on hypertext writing;
Johnson-Eilola, Selber, & Selfe, 1999 on the interface of computer technologies and technical communication; Sullivan & Porter, 1993 on writing technologies), the need to expand the foci of our studies remains. Johnson-Eilola, Selber, and Selfe urge researchers “to think critically, multiply, historically, and contextually about the ways computer technologies are developed” (p. 198) to understand the role of computers in our field.

Even those researchers who have turned a critical eye to software have not undertaken a broad study of the discourse of software and the software programming culture that is primarily responsible for both the production of software and its discursive coding. Such a study answers calls by both Selber and Johnson-Eilola et al. to study the practices and contexts in which computer technologies are developed. Additionally, this study responds to the urging of new media theorists to unveil the social and political implications of software by undertaking what has been coined “software theory” by Manovich (2001) and “software criticism” by Fuller (2003). Manovich argues that to understand new media necessitates an understanding of computer science and the means by which traditional media forms have been transformed through the digital. Likewise, Fuller contends that only by understanding software as culture can we locate the “implicit politics” of the technologies that increasingly organizes our everyday lives. By taking a discursive approach to software and the culture of software programming, I seek to contribute to the theory and criticism of software through expanding the avenues by which critical digital literacy may occur.

Cultivating critical digital literacy and the meaningful action that can result from it necessitates understanding social practices in context at the broad, macro-level social system of the software programming culture and at the more specific, localized micro-level situations in which cultural practices are enacted. The importance of context to studies of computer technologies leads me to the second need that my research addresses—the continued need to develop theories, methodologies, and frameworks that encourage a complex understanding of context.

To study social practices “in context” is not an easy task, a fact that prompted Schryer (2002), in her examination of the ideology manifest through the genre of negative letters, to state, “at this time I believe methodological and theoretical models are needed to allow…theorists to account for contextual and textual practices in a more critical way” (p.
Taking into account the matrices of forces that shape a given context—elements that occur at both a macro- and micro-level—is a necessary yet complex enterprise given that “context” as a strategic term can pose theoretical problems of its own, particularly in studies of technologies. Slack (1989) warns,

To assert that one studies technologies in context is really to take a particular position on what constitutes context and thus to enter a terrain where there is little agreement…More often than not, context is invoked as a sort of magical term, as if by claiming to take context into consideration, one could banish the theoretical problem of its specificity. (p. 329)

In a similar critique, Grossberg (1992b) argues that any study of social practices and their significance is in essence a study of cultural context. As such, these studies “cannot be a matter of merely acknowledging context…Too much of contemporary theory treats contexts as the beginning of analysis, as background which exists independently of the practice being studied and which can therefore be taken for granted” (p. 55).

Consequently, my own examination of the discursive coding of software is also by necessity a study of context and how we can understand the culture of software programming at a macro-level context that, though temporarily stabilized, is full of difference because of action at the micro-level. By emphasizing that a critical study of the practices of the software programming culture is also a study of context, I hope to avoid the pitfalls that both Slack and Grossberg claim are inherent to studies of social practices in cultural contexts.

I am particularly interested in the ways in which culturally-contextualized social practices are structured and, yet, are also the means by which actors affect change, a topic that has interested others in our field (Faber, 2002; Bazerman, 2002; Herndl, 1996; Miller, 1994). Bazerman, in his historical analysis of the “symbolic engineering” of incandescent light technology, notes that crucial elements in the success of the electric light as a marker of technological change were the discursive practices by which “stable meanings” were created within communicative systems (p. 335). In order to change the landscape of technology, Edison and his associates had to create stability of meaning in and through the discourse of incandescence. In another discussion of stability and change, Faber suggests that the motivation to bring about ideological change is rooted in efforts to introduce discursive
instability into the relatively stabilized context of higher education. This task is accomplished, according to Faber, through the “strategic ambiguity” created in the texts of “corporate universities” that seek to change the current humanistic ideology of higher education by introducing a “market-based ideology” into this context (p. 413). As these discussions evidence, stability and change are often intertwined and, as such, a critical study of stability is intricately tied to a critical study of change. As I have already suggested, understanding this tie demands examining social practices in cultural contexts at both the macro- and micro-levels, a move that foregrounds context rather than treating it as the background to a study.

By foregrounding that which has typically been relegated to the background— the relationship between the cultural macro-level context of the broader social system and the micro-level situation in which practices occur—I believe we can better understand the role of the practices of the software programming culture in ideologically encoding software. In doing so, we can decode the ideological significance of our own software use and determine whether or not that use is aligned with the politics and ideologies to which we consciously or unconsciously subscribe. Consequently, it is the process of signification through the discursive coding of software—the intervention of language into ideology (Grossberg, 1996a, p. 137)—by the software programming culture that occupies my research.

**Research Questions**

To address gaps in the critical research on computer technologies—in particular the need to study critically the contexts in which software, as technological artifacts, is developed and the need to develop theories, methodologies, and frameworks that encourage a complex understanding of context—I seek in this dissertation to answer the following research questions:

- What are the political ideologies encoded in software?
- What are the social practices by which these ideologies function in the software programming culture?
- What are the means by which laypersons can develop a critical literacy of software?
My answers will be partial, tentative, and contingent. Nonetheless, my analyses here contribute to critical research on computer technologies by mapping the cultural practices by which software is discursively coded. I also attempt to add to critical research on context that seeks to understand the relationship of stability to change. By doing so, I hope to provide an avenue by which lay users of software can foster digital literacy so that together we can challenge Feenberg’s assertion that “despite occasional resistance, the design of technical institutions disqualifies modern men and women from meaningful political participation” (p. 101). Through understanding of the shared cultural practices of software programming culture and the different ways in which they are enacted, I believe we can uncover the ideologies that are embedded in the software technology we use and identify the ways in which ideologies function through our everyday computer use. Only by doing so can we foster a digital literacy that allows every one of us, regardless of technical expertise, to participate in the discursive coding of software by which we are materially and ideologically encoded as digital subjects.

To this end, in the remainder of this chapter I define my use of the terms “discourse” and “culture” that inform the theoretical assumptions in my analysis of the discourse of the software programming culture. To conclude the chapter, I provide a brief overview of the chapters that follow.

THE RELATIONSHIP BETWEEN DISCOURSE AND CULTURE

“Discourse” and “culture” are perhaps two of the most examined and, as a result, most amorphous words in critical social theory. To make clear my own use of these terms, in this section I locate my usage of “discourse” in Foucault’s theory of discourse and “culture” in critical social theories that take a dialectical approach to social life.

Discourse

Like Foucault and Hall, I am interested in understanding the discourse of a particular formation, in this case the software programming culture. I focus on the ways that a “verbal network” (Foucault) of signifying practices structure this formation to produce regularity and continuity while at the same time creating opportunities for change and discontinuity in the circulation of power and ideology that results in and from the discursive coding of software.
Postmodern approaches to discourse are informed to a great degree by Foucault who refers to “discourse” not just as language but as the whole of social practices that are necessary to and dependent upon discourse for meaning and significance. As Foucault states, “nothing has any meaning outside of discourse” (1972, p. 37). Comprised of groups of statements, discourse encodes meaning on practices and objects within “discursive formations” (p. 38). Discursive formations, which occur “whenever, between objects, types of statement, concepts, or thematic choices, one can define a regularity...,” are structured according to “rules of formation” that shape and are shaped by the “practices that systematically inform the objects of which they speak” (p. 49). It is through this process that some practices become particular, appropriate, or natural to certain discursive formations while other practices do not. Together, statements within a particular discursive formation, in their similarities and differences, create a discourse, as Foucault’s own research demonstrates about the discursive formations of medicine (1974), punishment (1979), and sexuality (1978).

I also rely on Hall’s (1997) definition of discourse: “what one says (language)” and what one says about “what one does (practice)” (p. 44). This view highlights the breadth of discourse as a signifying network of language and practice. For both Hall and Foucault, the question that underlies analyses of discourse is not “what is actually being said in that which was said,” but “what is this specific existence that emerges from what is said and nowhere else?” (Foucault, 1972, p. 40, 29). Answering this question about any discursive formation requires an understanding of the network of signifying practices that mediates it; for, it is this network, composed of statements bordered by other statements, that does, in fact, construct context and the effects on statements possible within it (Foucault, 1972, pp. 97-98). In other words, statements do not exist in a vacuum, but instead are contextualized by other statements; together, these statements within a discourse create a context. Consequently, only through the discourse—what a group says about an object and the practices related to it—that encoding, and consequently meaning, occurs.

To illustrate, Foucault’s archeology of the discourse of punishment uncovers the levels at which statements, events, practices, and situations are possible (1979, p. 31) in the context of state-sponsored discipline. He traces the technologies of punishment in France
from approximately 1757 and the death by public torture of Damiens, a would-be assassin of King Louis XV, to 1840 and the opening of Mettray, a “cloister, prison, school, regiment” that heralded the modern social practices of disciplinary punishment. Foucault demonstrates that these technologies of punishment produced and were produced by particular techniques, or what are more commonly referred to now as social practices. In essence, Foucault explores the discursive encoding of punishment and the practices through which this encoding occurred by examining the “rules of formation” of the network of statements that made possible and internalizable—encoded—practices of punishment. Perhaps the most often-cited mechanism that Foucault discusses is the Panopticon—Bentham’s architectural design in which a “supervisor” can always observe from his tower “a madman, a patient, a condemned man, a worker or a schoolboy” who resides in the cells in the building ring around the tower. The design of the Panopticon promotes particular disciplining practices across institutions ranging from prisons and hospitals to factories and schools. This “diagram of a mechanism of power” (p. 205) induces a variety of institutionally structured social practices (i.e., practices that are meant to “…reform prisoners…treat patients, to instruct school children…” (p. 200). As Foucault explains, through the Panopticon, modern society, as only the modern society could, produced and reproduced a disciplining mechanism that promoted practices of surveillance—both real and imagined—as the major mode of punishment. Because instantiations of these institutional “apparatuses” and the practices that structure and are structured by them proliferated and migrated to every facet of modern living, the Panopticon and the practices it facilitates function as a disciplining mechanism of “everyday life” (p. 205).

Foucault’s study of the discourse of punishment does not consider the struggle revealed through the archeology of a discourse (and its verbal network through which power is negotiated) as something inherently negative. Because power is not simply something some have to the detriment of others, power must be understood as inherently generative because it is through discourse that meaning is made. As Foucault argues, “We must cease once and for all to describe the effects of power in negative terms: it “excludes”, it “represses”, it “censors”, it “abstract”, it “masks”, it “conceals”. In fact power produces; it produces reality; it produces domains of objects and rituals of truth” (1979, p. 194). As such,
power, in the case of the Panopticon, does not reside in the Panopticon itself or in the “supervisor” in the tower; instead, power circulates through the statements about the Panopticon as well as through statements by and about the supervisor and prisoner alike. Foucault (1980) writes,

\[
\text{Power must be analysed as something which circulates, or rather as something which only functions in the form of a chain. It is never localized here or there, never in anybody’s hands, never appropriated as a commodity or piece of wealth. Power is employed and exercised through a net-like organization. And not only do individuals circulate between its threads; they are always in the position of simultaneously undergoing and exercising this power. In other words, individuals are the vehicles of power, not its point of application. (1980, p. 98)}\]

In the mechanism of the Panopticon, supervisors, for instance, in their exercise of power through practices of surveillance are also subject to that same power—surveillance by others. Anyone coming into the Panopticon can easily assess how well the supervisor is doing because the whole building, its inmates, and functions are open for observation (1979, p. 204). If the inmates act or speak out contrary to the rules and regulations, they are able to employ power, the significance of which is, of course, dependent upon situational factors such as the stability of meaning in what is being said as well as who is speaking and to whom. Consequently, in this Foucaultian paradigm, power is relational rather than attributive, a fact that allows for disruption, struggle, and change. By accounting for struggle as a result of the “intervention of ideology into language” (Grossberg, 1996a, p.137), as Hall does, we can understand the ways in which ideology and power results in the “multiplicity of force relations” (Foucault, 1979, p. 92) through discourse. Discourse then is not only the means by which power is wielded but is power itself; “discourse is not simply that which translates struggles or systems of domination, but is the thing for which and by which there is struggle, discourse is the power which is to be seized” (1984, p. 110).

In discursive studies within professional communication, researchers have been particularly interested in Foucaultian notions of discourse because of its view of language, practice, and power as intertwined (see Herndl, 1996; Ranney, 2000; Sheehy, 2003; Stygall, 1994; Wilder, 2005). Both Ranney and Herndl emphasize the role of discursive practices in
both the reification of and resistance to dominant ideologies. In the case of the Ranney’s study of the national sexual harassment policy, individuals’ participation in the development of a “culture of discussion” in the workplace offers the means by which to resist through discourse the reality created through the language of the policy, language which (re)inscribes an ideology that leaves individuals with little to no recourse for action. Similarly, Herndl’s analysis of a civilian biologist’s resistance through the discursive practice of writing to the ideology of efficiency promoted by the military base where he works demonstrates the relational nature of power. An attributive view of power would inscribe the military complex with all the power and a lone biologist who resists dominant institutional practices with no power. However, the tactics of the biologist demonstrate the ways in which an individual can exercise power because of the relational nature of power. Power is not located solely in the sexual harassment policy or in the military institution. Actors are able through their discursive practices to assert agency even in contexts that might otherwise seem to be the singular determinant of actors’ actions.

Like Foucault in his study of the verbal network of punishment, I analyze the verbal network of the software programming culture in terms of both what actors say about software and what actors say about what they do with software. Beginning with January 1975—the year that the Altair 8800 microcomputer, the first generation personal computer (PC), appeared on the cover of Popular Electronics—and ending with the December 2005—the time by which the SCO v. IBM was to be decided (though it was not and has yet to be)—I review what other researchers interested in the practices and history of software have said about software. I also describe and analyze what those primarily responsible for the development and regulation of software—software programmers—have said about software and what they do with software. Finally I consider what those who have come to be included in the daily administration of software—corporate executives, lawyers, watchdog organizations—have said about software and what they say about what they do with software. I use these accounts to map the verbal network of this discursive formation that reveals the struggle to induce stability of meaning to software, software programming, and the software programming culture itself. In drawing on statements regarding a variety of aspects of the culture, from development processes and software licensing agreements to organizational descriptions and
stories of software, I seek to understand how those who speak of the same thing—software—
can speak of it so differently and how this difference has significant ideological implications
for lay users.

Similar to studies of workplace organizations by researchers such as Ranney and
Herndl, my analysis of the ways in which power circulates through the discourse of software
offers an opportunity to illuminate the discourse by which cultural practices of the software
programming culture are encoded and the means by which actors are limited and empowered.
By focusing on the discursive coding of software, I, as a lay user of software, seek to
understand how meaning is encoded for and through software, how ideology and power
circulates through the discourse of software, and how this discourse is reflexively employed
by actors to stabilize, align, and reproduce meaning as well as to challenge, struggle, and
transform meaning. But to construct this verbal network of software so as to analyze the
statements that are both contextualized by and produced by this network, I analyze the
discourse of the software programming culture, the discursive formation within which these
statements are produced. In the following section, I explain how understanding software
programming as culture allows me to narrow my study to a discursive formation constituted
by a set of practices, including, but not limited to, software programming.

Culture

Just as important as the concept of “discourse” to my study is the concept of
“culture,” a term that is contested and, in many cases, problematic. My own use of “culture”
is very much dependent upon Foucault’s and Hall’s discussion of discourse and the ways in
which social practices within a discursive formation create regularities through shared
cultural practices. The set of shared cultural practices I examine differentiates the software
programming culture from other cultures. At the same time, these shared cultural practices
provide the means by which difference within a culture can occur.

Because a culture, especially in the networked society of the Information Age, can be
dispersed throughout the world via the Internet, I do not locate culture in specific
geographies and physical locations, an approach that differs from many identifications of
culture (see Gray, 1999). Longo (1998, 2000), who warns against locating culture within
traditional notions of workplace organizations that traditionally are set in the same
geographical location, notes that geographic identifications of culture can limit the possibility of understanding the complex web of assumptions and contestations that influence and are influenced by a particular organization. This geographical approach to the software programming culture has traditionally been a popular one (see Hales, 1995; Carmel, 1997; Raijlich, Wilde, Buckellew, & Page, 2001; Wiegers, 1996). Creating a software engineering culture by Wiegers, a software process consultant, offers an account of a number of software development teams at Eastman Kodak. Motivated by a desire to help organizations create an atmosphere for quality software production, Wiegers describes a successful framework through which an “organization grows a quality-directed software culture by blending established approaches from many sources with locally developed solutions to specialized problems” (p. 4). In “Software cultures and evolutions,” computer scientists Raijlich, Wilde, Buckellew, and Page explore the different stages in software development and use in order to understand “legacy,” or outdated, software that continues to be in use. Raijlich et al. argue that “software engineers need to understand a legacy computer program's culture—the combination of the programmer's background, the hardware environment and the programming techniques that guided its creation” (p. 28). Kling and Carmel, in “American hegemony in packaged software trade and the ‘culture of software’,” examine the cultural reasons for the U.S.’s dominance. They argue this dominance is caused by the U.S.’s “culture of software,” which values individualism, entrepreneurialism, and innovation. In these discussions, software culture is located primarily in geographical location: the organization of Eastman-Kodak, the specific workplaces of software engineers, and the national boundaries of the U.O.

Other studies of the software programming culture have also moved beyond geography to the kind of cultural approach rooted in discourse that I use for my study of the discourse of software (see Baym 1995; Elliot & Scacchi, 2003). For instance, in “From practice to culture on Usenet,” Baym, a sociologist, describes how discussions using Usenet, a pre-Internet computer communication network through which individual users post and read messages in topic-oriented newsgroups, “can operate as a culture-creating force” (p. 29). From her ethnographic study of a Usenet group that discussed daytime soap operas, Baym concludes that, through their discursive practices, members reproduce the functional structure
of Usenet by “marking messages” with specific features (e.g., headers, quotation systems, signature file). At the same time, the members create a community with a specific identity by “transforming a new medium into something unforeseen by its creators” (p.51). Elliot and Scacchi, of the Institute of Software Research at the University of California, Irvine, report on practices of teamwork, community building, and conflict resolution in a virtual community dedicated to the development of the free software project GNUe. The everyday practices, or “norms of the culture,” are a result of the values and beliefs that members of this culture aligned themselves with, according to Elliot and Scacci. Software programmer and self-described “historian and resident ethnographer” of the Open-Source community, Eric Raymond (1999), has also described the social practices of “hacker culture” that, in the era of the PC, led to open-source software and the community responsible for its development.

“Culture” is such a problematic term that Raymond Williams (1985) wished that he had never heard of the “damned word” (p. 154). But the ambiguity of the term, noted by a number of critical social theorists (Archer, 1996; Bell, 1998; Bourdieu, 1984a; Hall, 1980; Hays, 1994; Williams, 1976), has led Hall, in his own examination of “culture,” to suggest that “no single unproblematic definition of ‘culture’ is to be found” (p. 58). One reason for the difficulty is Cartesian dualism (e.g., mind/body, individual/society, macro/micro, structure/agency), which has marked traditional studies of social systems such as culture. In Cartesian formulations, culture is either an objectivist or subjectivist phenomena—all macro-level objective structures or all micro-level subjective acts of agency. Hays (1994) suggests that such dualistic thinking has led to what she succinctly calls the “sticky problem of culture” (p. 65). She explains,

Culture is sometimes reduced to an epiphenomenal expression of the mode of production, the relations of production, or the relations between states and classes; is sometimes treated as a “mere” ideological legitimation of the material intersects of profit maximizers; at other times is regarded as an insignificant, private, internal subjective reflection of the public, external, objective world; or is sometimes reduced to “soft,” infinitely malleable, “free-floating” ideas. All of these common and often unexamined usages are misleading and should be abandoned.
Hays, like many other critical social theorists, calls for abandoning notions of culture wherein culture is understood either as an objective force that structures reality through macro-level phenomena, such as the economy, or as the subjective micro-level experience of the individual. The approach that I adopt to avoid this dualistic quagmire is to consider culture as it is constituted within discourse.

Consequently, rather than locating culture in a shared geographical site that provides the macro-level context for situating social practices, I locate the site of my analysis in the written discourse of members of a culture and, like Foucault, within a particular historical time period. Situating culture in the discourse of members of the software programming culture enables me to emphasize the constitution of the culture through its shared discursive practices and to analyze the means by which stability and change occur at both the macro- and micro-levels.

Culture in this perspective becomes, in the words of Hall (1997), “not so much a set of things…as a process, a set of practices. Primarily, culture is concerned with the production and the exchange of meanings—the giving and taking of meaning—between members of a society or group” (p. 2). In this approach, culture is viewed as a relatively bounded social system organized around a particular set of practices and constituted by discourse that gives meaning to those practices as well as to the culture itself. To analyze culture amounts, then, to the “clarification of the meanings and values implicit and explicit in particular ways of life, a particular ‘culture’” (Williams, 1961, p. 57). By examining the discourse of software produced over a 30-year period, I begin to unpack the ways in which what is said and what is said about what is done stand as negotiations of power and ideology and serve as the discursive formation of the software programming culture. Additionally, such a study facilitates the building of context of software programming at both the macro- and micro-level. This attention to cultural context works to uncover the relational ways through which these negotiations of power are efforts to both stabilize and change the meaning of software through discourse.

In my analysis of the software programming culture, I build on this cultural-practice approach to software to identify and describe the set of shared cultural practices that function as the stabilizing structure of the software programming culture. However, to resist a purely
structuralist account of culture, I analyze how these shared practices are encoded differently by different actors who influence the culture. Doing so allows me, in the words of Foucault (1972), “to show what the differences consisted of, how it was possible for [individuals] within the same discursive practice, to speak of different objects, to have contrary opinions, and to make contradictory choices” (p. 200).

**Overview of Chapters**

My aim in the following chapters is to create temporarily stabilized answers to my research questions, which, again, are:

- What are the political ideologies encoded in software?
- What are the social practices by which these ideologies function in the software programming culture?
- What are the means by which lay persons can develop a digital literacy of software?

In Chapter Two, I begin by reviewing theories of practice that have added to our understanding of how both stability and change within social systems occur through social practices, discourse in particular. Included in my review are theories that have already proven useful to discursive studies: Giddens’ structuration theory, cultural studies’ articulation theory, and Bourdieu’s constructivist structuralism. However, in an effort in to avoid a potentially unreflexive adoption of these theories of practice for my study of the discursive coding of software, I also review critiques of these theories. These critiques challenge the usefulness of theories in overcoming the dualism in which either stabilizing structures or dynamic agency is privileged. This is the very dualism that theories of practice seek to escape. In light of these critiques, I avoid using any one theory of practice and instead draw from all three theoretical perspectives to discuss the relationship of structure and agency. I also borrow from sociologists Emirbayer and Mische’s work on time as “temporal orientations” and postmodern mapping’s location of space as social positionings to build an analytical framework that I term “the spatio-temporal framework.” Finally, I describe my methods of data gathering and explain my choice of methods, which are guided by the spatio-temporal framework.
In Chapter Three, I begin to answer the question: What are the social practices by which ideologies function in the software programming culture? I identify and describe the shared typified social practices of the software programming culture—in short, the cultural practices—that provide stability to the culture and its members. I argue that doing so allows lay users to begin to understand the practices by which ideologies function discursively in the everyday activities of members of this culture and, as a result of the pervasiveness of software, in the everyday use of software by lay users. Awareness of these iterative practices and the ways in which they are routinely “reactivated” (Emirbayer & Mische, p. 975) is a first step through which lay users can become familiar with the software programming culture, and, in doing so, identify those already naturalized practices that provide avenues for their potential participation in the discursive coding of software. Such participation affords lay users opportunities to develop their own critical digital literacy and increases the possibility of the effective exercise of power through discourse.

In Chapters Four and Five, I answer the question: What are the political ideologies encoded in software? Having identified and described in the previous chapter a set of typified cultural practices that members of the software programming culture reactivate through their everyday routines, I now turn to the study of the encoding of software beyond its functional meaning through the discursive coding of software. Through an analysis of the narratives of different communities and their spokespersons, I emphasize the ideological encoding of software and its related practices and examine in detail the ways in which these actors, and the communities to which they belong, imagine ideologically-steeped trajectories for future action that create alternatives to relatively stabilized meanings of software development and use. To this end, in Chapter Four I analyze the encoding of software through the earliest archived narratives of members of the software programming culture, members who were to become spokesperson for yet-to-form communities within the culture.

But, because issues of ideology involve collectives rather than individuals (Hall, 1996), I examine in Chapter Five how communities that correspond to particular identities of the network society (Castells, 1997) have emerged that both sanction and reproduce the discursive coding of software constructed through the narratives of these communities’ spokespersons. I uncover the struggle within the software programming culture to encode
software and the resulting plurality of discursive coding that occurs. Analyzing the narratives of software that circulate in the software programming culture allows lay users of software to begin to see the ways in which software is ideologically encoded, the ways in which through their software use is ideological, and the ways in which they are encoded as digital subjects.

In Chapter Six, I return to the question for which I began to formulate an answer in Chapter Three: What are the practices by which ideologies function in the software programming culture? After identifying and describing the typified cultural practices of the software programming culture (Chapter Three), and analyzing the ways in which software and its related-practices are encoded differently by the competing communities (Chapters Four and Five), I conduct a close analysis of three copyright licensing agreements and locate the different applications of software licensing agreements as the result of deliberative action. Each of these agreements is associated with a community I discuss in Chapter Five. My analysis suggests the ways in which the regulatory practice of copyright in the form of software licensing agreements functions as both “medium” and “resource” (Giddens 1984; Miller, 1994b) to control the copying, distribution, and modification of software. In short, software licensing agreements are a typified medium that structures the ways in which actors regulate and are regulated in their use of software. At the same time, these licensing agreements are a resource through which actors, in both their development and use of software, can strategically enact their ideological beliefs and have those beliefs enacted on others. Consequently, lay users can understand the ways in which their own software use is ideologically regulated. Lay users of software can also begin to think about the ways in which their software use is or is not aligned with the identity and ideologies they align themselves with or against.

Finally, in Chapter Seven, I explicitly address my final research question: What are the means by which laypersons can develop a digital literacy of software? In doing so, I discuss how understanding the practices by which ideologies function in the software programming culture and the ideologies that are encoded in software offer opportunities by which the lay users of software can enact their own digital literacy.
CHAPTER TWO: THEORY, FRAMEWORK, AND METHODS

In this chapter, I describe the theory, framework, and methods that inform and guide my analysis of the discursive coding of software by the software programming culture. I begin reviewing theories of practice, which include Giddens’ structuration theory, cultural studies’ articulation theory, and Bourdieu’s constructivist structuralism. These theories of practice reflect my own belief in the importance of resisting the tendency to privilege either an objectivist or subjectivist understanding of reality. To resist this dualism, theories of practice argue that reality occurs through the interplay of structure and agency within social practices. Because discourse—“what one says (language)” and what one says about “what one does (practice)” (Hall, 1997, p. 44)—constitutes and is constituted by the meaning encoded by and through social practices, I use these theories to inform my methodology.

In an effort to be reflexive about my own methodological viewpoint and understandings of the ways in which reality is created through discourse, I also review critiques of these theories of practice. The work of Giddens, of theorists in cultural studies, and of Bourdieu is influential in studies of discourse within the field of rhetoric; however, our field has yet to sufficiently explore critiques of these theories, most of which arise out of the social sciences. Though applications of these theories of practice have generated much productive research in our own field, the critiques led me to think through the development of a framework that I could usefully apply while avoiding some of the important problems that the critiques raise.

The analytic framework that I apply, which I call the spatio-temporal framework, is rooted in the belief that space and time are important to understanding the relationship between stability and change in social practices. To build this framework, I borrow heavily from sociologists Emirbayer and Mische’s (1998) work on the temporality of agency and from discussions of space in postmodern mapping research. I believe that in its emphasis on space and time, the spatio-temporal framework complements theories of practice by providing a guide to choosing methods of analysis that help to uncover the ways in which social practices are sources of both stability and change within a particular social formation, in the case of my study, the software programming culture.
Finally, I describe my choices of texts for analyses from this culture and my choice of methods for analyzing those texts. Both of these choices were guided by the spatio-temporal framework and enable me to uncover the relationship between stability and change, as well as the reality created through the discourse of the software programming culture in its discursive coding of software.

**Theories of Practice**

Hay (1995), a sociologist interested in the ways in which stability and change occur within a social system, asserts that “every time we construct, however tentatively, a notion of social, political, or economic causality, we appeal, whether explicitly or (more likely) implicitly, to ideas of structure and agency” (p. 198). Theories of practice are key to understanding such appeals because these theories seek to explain how causality occurs without falling into explanations that reify worldviews that are fundamentally flawed because they are either objectively or subjectively determined. In fact, theories of practice are notable in that they embrace a postmodern eschewing of positivism and at the same time respond to subsequent critiques of postmodernism. Whereas positivism argues that social life can be understood through scientific rigor and “systematic, controlled, empirical” study that reveals an objective reality free of the influence of social contexts (Kerlinger, p. 10), postmodernism argues that social life can only be understood as an endless game of discursive play. Yet while postmodernism has certainly triumphed in humanities research, the inability of postmodernism to explain how deliberate strategic action can catalyze change has undermined its potential. As Faigley (1993) explains,

> Postmodern theory has not produced, however, a broad theory of agency that would lead directly from these critiques to political action. Indeed, the incisive critique in much of postmodern theory is inimical to such efforts, viewing them as a way of closing off critique too quickly and short-circuiting its radical potential, even replacing old structures of domination with new ones. (p. 39)

In a similar, though more foreboding critique, Bordo (1999) argues that postmodernism has in fact already replaced an old structure with another and in so doing has reified the very Cartesian dualisms, including objectivism/subjectivism, that postmodernism sought to thwart. Bordo blames this reification on a “theoretical hubris” on the part of those who
embrace a postmodern perspective of the world. This hubris privileges those who adopt a postmodern perspective and renders those who don’t as “Others” who can not see the world, let alone act in it (pp. 279-80). Consequently, for Bordo, “the question remains, however, how the human knower is to negotiate this infinitely perspectival, destabilized world” (p. 226).

Theories of practice that seek to answer Bordo’s question include Giddens’ structuration theory, cultural studies’ articulation theory, and Bourdieu’s constructivist structuralism. In this section, I review how these theories of practice construct a way out of the double bind described by Faigley and Bordo and provide an avenue for methodological approaches such as the one I apply to the study of the discursive coding of software by the software programming culture. Discussing the relationship between theory, methodology, and method, Zuboff (1988) writes,

> Behind every method lies a belief. Researchers must have a theory of reality and of how the reality must surrender itself to their knowledge-seeking efforts. Each epistemology implies a set of methods uniquely suited to it, and these methods will render the qualities of data that reflect a researcher’s assessment of what is vital. (p. 423).

In short, the theories of practice that I discuss here have, in part, shaped my own beliefs about how reality works and the ways in which research can explain those workings. Consequently, in reviewing these theories of practice, I explain the theoretical lens that informs my methodology, which, in turn, informs my methods of analysis.

**Structuration Theory**

I begin by discussing Giddens’ theory of practice, structuration theory, which takes as its primary analytic concept “duality of structure.” According to Giddens, structuration theory resists the “dualism associated with structure and agency” (1984, p. xxvii), emphasizing instead the “mutual dependence of structure and agency” (1979, p. 69). Through the lens of structuration theory, the object of study in the social sciences should be “neither the experience of the individual actor, nor the existence of any social form of societal totality, but social practices ordered across space and time” (p. 69). As such, social systems such as cultures are understood as recursive rather than as linearly progressive (p. 2).
because, according to Giddens, “the structural properties of social systems are both the medium and outcome of the practices they recursively organize” (1984, p. 25). Pre-given social structures and the actions of actors are thus understood not as independent from one another but instead as interdependent—the one can not exist without the other: “That is to say, [structures] are not brought into being by social actors but are continually recreated by them via the very means whereby they express themselves as actors” (p. 2). Consequently, we are able to speak of structures as distinct from, but never independent of, actors and vice versa.

According to Giddens, actors recreate social structures through their knowledge of the rules of that structure, what Foucault would perhaps call the “rules of formation.” Because these rules function as “resources” and are “inherently transformational” (Giddens, 1984, p. 17), the assumption is that actors, inasmuch as they recognize and understand these rules, are able to “know a great deal about what they are doing in the processes of interaction…” (1979, p. 215-216). But, because the ways in which structure and agency play out in a given situation within a social system are typically not always readily apparent, “…there is a great deal which [individuals] do not know about the conditions and consequences of their activities, but which none the less influence their course” (p. 216). In articulating the duality of structure, Giddens puts forth a perspective that challenges the objectivism/subjectivism dualism and considers structure and agency as intertwined in the construction of social life at both the macro- and micro-level. By uncovering the interplay between structure and agency, researchers can render transparent the ways in which actors becomes agents, whom Giddens defines as those who “could have done otherwise” (1995, p. 63).

Giddens’ structuration theory has influenced a number of researchers in a number of disciplines, including our own (see Bazerman, 2002; Faber, 1999, 2002; Herndl, 1993, 1996; Johnson-Eilola et al, 1999; Miller 1994b; Schryer et al., 2002). The lens of structuration theory has provided significant insights into the ways in which stabilized structures are recreated through the discursive actions of actors. Herndl, in one of the earliest uses of Giddens’ structuration theory in rhetorical research, suggests that the duality-of-structure approach offers an alternative to singularly descriptive writing research and pedagogy by uncovering the institutional ideologies of both academic and non-academic workplaces and
how power circulates through them. Uncovering these ideologies is, according to Herndl, a process that helps students become “self-reflexive agents” (p. 360). Bazerman frames the story of Edison and the electric light in structuration theory to emphasize Edison’s creative recreation of existing social structures and the effect of Edison’s action in changing the traditional meaning of technology by stabilizing the meaning of incandescence. Similarly, Miller’s use of structuration theory calls for the (re)envisioning of genre as a structure that rather than simply limiting rhetorical strategies provides resources for writers who, in turn, (re)create structure through their practice of genre-in-action. Genre consequently becomes “both means and end, both resource and product” (p. 70), a characterization that speaks exactly to Giddens’ duality of structure.

**Articulation Theory**

Another significant theory of practice is articulation theory. Articulation theory is cultural studies’ response to the objectivism/subjectivism dualism that was manifest in its own development as a metadiscipline. In its early formations, cultural studies was closely aligned with cultural humanism, especially in the works of Raymond Williams (Grossberg, 1996a). In response to this “culturalist” approach that emphasized individual subjectivity, a structuralist approach, influenced by the works of Marx and Althusser, arose. Social research taking this approach emphasizes the effects of the structural system of capitalism on individuals. In his historical overview of cultural studies, Hall (1980) describes the relationship between the culturalist and structuralist approach:

> We can identify this counterposition at one of its sharpest points precisely around the concept of “experience”, and the role the term played in each perspective. Whereas, in “culturalism”, experience was the ground—the terrain of “the lived”—where consciousness and conditions intersected, structuralism insisted that “experience” could not, by definition, be the ground of anything, since one could only “live” and experience one’s condition *in and through* the categories, classifications and frameworks of culture. (p. 66)

To combat these dualism-reinforcing approaches to the study of reality—the one emphasizing subjectivity, the other structuralism—researchers in cultural studies have embraced articulation theory.
Articulation theory advocates a non-reductive approach to understanding social practices in context and assumes a non-essentialist approach to the encoding of meaning. Meaning is not determined by either social structures or individual experience but rather in the ways in which certain forces are brought into relation through the interplay of structure and agency at historically-specific moments (Grossberg, 1992, 1996a; Hall, 1980; Slack, 1996). Articulation theory encourages the realization of the “non-necessary link” between meaning and practices in radically contextualized sites of knowledge production (Slack, p. 117). According to Slack, to understand how meaning is articulated means to understand “how ideological elements come, under certain conditions, to cohere together within a discourse, and a ways of asking how they do or do not become articulated, at specific conjunctures, to certain political subjects” (p. 117)

Researchers in cultural studies often utilize the “circuit of culture” or elements from this circuit to position forces in a cyclical process of production, consumption, representation, identification, and regulation (du Gay et al., 1997). This positioning, according to proponents, offers a way to escape an overly structuralist (i.e., economically deterministic) or culturalist (i.e., subjectivist) understanding of social life so as to “historicize, contextualize, and pluralize by highlighting the contingent, provisional, variable, tentative, shifting, and changing” (West, p. 11). Consequently, Slack (1996) argues that context must be a focus of study because “context is not something out there, within which practices occur or which influences the development of practices. Rather, identities, practices, and effects generally, constitute the very context within which they are practices, identities or effects” (p. 125). In that sense, to “map the context” becomes an object of study in and of itself. Only by mapping production of both stability and change through discourse is it possible for researchers to understand social life, and, potentially, for actors to rearticulate meaning through acts of resistance and political action. These are the aims of cultural studies’ articulation theory.

Articulation theory has been useful to a number of researchers in rhetoric (see Geisler, 2003; Gurak, 2002; Johnson-Eilola, 1997; Slack, Miller, & Doak, 1993; Wilson, 2000). For example, Slack et al. advocate the articulation mode of communication, which (re)articulates the position of the technical communicator from subordination to
empowerment. By understanding meaning as dialectically constructed among the sender, receiver, and the mode of transmission, the technical communicator is constructed as essential to successful communication practices (p. 27). Geisler has also used articulation theory to analyze the rhetoric of tech reviews. Refuting the concept of a “simple technology transfer model,” in which tech reviews are “part of a one-way transfer of technology from point of production to point of consumption” (p. 27), Geisler uses cultural studies’ circuit of culture to demonstrate the dynamic relationship between production and consumption. The genre of tech reviews in this approach does not seek merely to sell a product like mobile technologies (i.e., cell phones, PDAs) or provide information to consumers about the product being reviewed. Instead, the tech review is imbued with what Geisler calls a “double voice…both speaking forward in the circuit of culture to help consumers evaluate new technology, and speaking backward to the producers to critique some paths of production and advocate others” (p. 27).

Like Slack et al and Geisler, Johnson-Eilola uses articulation theory to frame a call for the (re)articulation of the genre of hypertext. Examining hypertext through the lens of articulation theory fosters, according to Johnson-Eilola, critical research practices that view hypertext as variable and open in meaning rather than as bounded by objectively structured disciplinary conventions. Articulation theory, which “offers a practical approach to remaking borders” (p. 43), encourages new technologies of composing such as hypertext to be understood as “activities of reading and writing [that] are transformed and appropriated by widely divergent communities, each of which reconstructs general characteristics of hypertext in relationship to that community’s goals” (p. 7).

**Constructivist Structuralism**

Another theorist who has had a significant impact on research about the roles of social practices in issues of stability and change is Bourdieu. Like Giddens and proponents of articulation theory, Bourdieu seeks to undermine the objectivist/subjectivist dualism through a reflexive, dialectical approach to culture that he calls “constructivist structuralism or structuralism constructivism [sic]” (1989, p. 14). The joining of these two potentially-loaded terms reflects Bourdieu’s view that reality is constituted within the interplay of structure and agency rather than by privileging one or the other by “thinking in couples” (1990, p. 80).
Through the lens of constructivist structuralism, social life is understood not as determined by either large-scale structures or by individual subjectivities but as mediated by the interplay of the two.

Bourdieu argues that social practices are shaped by the structures of a particular field—“a network, or a configuration, of objective relations between positions” (Bourdieu and Wacquant, 1992, p. 97)—and habitus—the “durable, transposable dispositions” (Bourdieu, 1990, p. 53)—of individuals within that field. Accordingly, constructivist structuralism encourages researchers to analyze the interplay between field and habitus. This focus enables researchers to “say that individuals make choices, so long as we do not forget that they do not choose the principles of these choices” (Bourdieu, 1990, p. 45). Situating his approach in empirical research ranging from the study of the Kabylian ritual of gift exchange (1977) to French bourgeois society (1984a), Bourdieu argues for the importance of reflexive post-positivistic research practices, what he calls a “theory of practice.” Theory of practice obliges [researchers] to realize that the two approaches, structuralism and constructivist (by which I mean a kind of phenomenology of one’s initial experience of the social world and of the contribution which this experience makes toward one’s construction of the world), are two complementary stages of the same procedure.

(Bourdieu 1984b, p. xiv)

Positioning structuralism and objectivism as complementary highlights the importance of the one to the other in understanding how meaning occurs through social practices.

The influence of constructivist structuralism is evident in a number of studies in our field (see Faber, 2002; Herndl & Nahrwold, 2000; Porter et al., 2000; Schryer, 2000, 2002; Sullivan & Porter, 1997). For example, in their investigation and promotion of critical research practices in computers and composition, Sullivan and Porter’s own research practices emphasize the role of habitus in their aim to subvert the objective/subjective dualism. They argue that research on social practices typically conceives of action through a “structuralist master plan” or overemphasizes subjectivity at the expense of “social knowledge” (p. 24). To combat this dualism, Sullivan and Porter argue for a “postmodern methodology” (p. 186) that recognizes the “situatedness” of rhetorical practice. Part of this methodology draws on Bourdieu’s concept of habitus, which they describe as “the repeated
rituals of a culture: that which is done repeatedly perhaps for no ‘reason’…other than that it is that which is done” (p. 24). Examining these repeated rituals paves the way for understanding the invested and contextualized ways in which actors act on an everyday basis as well as the heterogeneity that can arise from these practices.

Herndl and Nahrwold have also borrowed the theoretical lens of constructivist structuralism include. Using Bourdieu’s theory of practice, they argue for the recognition of the “changing circumstances of a discipline” and the way in which these changing circumstances render research as “both regulated and open to variation and change” (pp. 266-267). Additionally, Faber borrows concepts from constructivist structuralism to frame his study of organizational change and the methods for studying such change. Two of these concepts include habitus and theory of practice. In analyzing the relationship between structure and agency as manifest in the narratives and identities created within organizations, Faber suggest, “habitus is useful on both practical and theoretical terms as it describes and explains how people integrate themselves into society through learned but unacknowledged daily activities, habits, and routines” (p. 141).

Through the import of Bourdieu’s constructivist structuralism, cultural studies’ articulation theory, and Giddens’ structuration theory, researchers have illustrated the importance of theories of practice for understanding how discourse and the social practices of both researchers and research participants are shaped by social structures (e.g., academic disciplines, workplace organizations, and genre) and the actions of actors who through their practices creatively reproduce this structure. Although structures and the practices these structures shape and are shaped by produce stability over time (this is what allows us to recognize social structures and systems as such), manifestations of agency including heterogeneity, action, and resistance, are all possible, as the researchers discussed in this section have illustrated.

**Critiques of Theories of Practice**

As the previous section illustrates, theories of practice have greatly influenced critical research practices in the field of rhetoric. However, critiques of these theories have not been as widely discussed. Just as Herndl and Nahrwold argue that descriptive research theories and practices have led to “theoretical imperialism” in the field of rhetoric (p. 289), I argue
that to avoid a similar imperialism through our use of theories of practice, we must reconcile the methodological usefulness of these theories with their critiques. By doing so, we can increase our methodological reflexivity and contribute to theories of practice very same recursivity that we seek to uncover in our studies of discourse. In brief, the critiques include

1. that structuration theory either reinforces the objectivism/subjectivism dualism or collapses structure and agency by failing to account for how structure and agency occur though social practices, 2. that cultural studies’ emphasis on practices of production and consumption often fails to account for the means by which actors can rearticulate their lives, and 3. that constructivist structuralism privileges objectivism through an articulation of habitus that leaves little possibility for change. I do not raise these social-science critiques to challenge either the theorists who developed these approaches or the researchers in our field who have borrowed these theories to frame their own studies, but rather to challenge my own investment in these approaches, a move that I believe is in keeping with the discussions of reflexive research practices put forth by these theories of practice.

Although Giddens’ structuration theory has been adopted widely, critics have charged that the duality of structure has fallen short of producing a dialectical approach to culture (see Archer, 1996; Hay, 1995; Sewell, 1992; Stones, 1991; Thrift, 1985). Instead of emphasizing social structures as both “medium and outcome of practices” (Giddens, 1984, p. 25) that also function as resources through which actors recreate those structures (p. 2), structuration theory is apt to reinforce the very objectivist/subjectivist dualism that Giddens sought to undermine. Thrift likens structuration theory to “all micro-situations and world empires with nothing much in between” (p. 434) Hay reaffirms this critique by asserting that Giddens’ detachment of everyday practices from larger social contexts leans toward intentionalism. The danger of intentionalism lies in its focus on “the micro-practices of social interaction as opposed to the macro-embeddedness of action within broader social and political structures” (p. 196). Action is thus divorced from the larger social structures that shape what ever form that action takes, and, as a consequence, the dualism of structure and agency is reified by privileging subjectivity.

In another critique of structuration theory, Archer likens Giddens’ formulation of structure and agency to a “conceptual vice” (p. 87) that elides objective structures and
agentic action, making them “mutually constitutive” (p. 76). Because of this mutual constitution, it becomes difficult to tease out the ways in which action is shaped by structure and agency because the two are collapsed within one another. As a result, structuration theory encourages “vague universal processes whose presence or absence is broadly associated with stability or change rather than being able to come to grips with the conditions and mechanisms involved” (p. 96). In a similar critique, McAnulla argues that in Giddens’ own use of structuration theory the research is “agency-centered…leaving structural or systems-based analysis for other work” (p. 280). This might not be surprising in light of Grossberg’s charge that too many studies of cultural practice treat the macro-level contexts of social practices (e.g., institutions, culture, society) “as background which exists independently of the practice being studied and which can therefore be taken for granted” (1992, p. 55).

Articulation theory has also received criticism for reifying rather than undermining dualistic thinking through its frequent reduction of culture, society, and the individual to issues of production and consumption (see Kellner, 1997; Story 1999; Grossberg, 1987). The emphasis on production and consumption, two elements of the circuit of culture, threatens to undermine the move toward radical contextualism by backgrounding the other elements in the circuit—representation, identification, and regulation. Because production in this dualism is governed by structure and “consumption is governed by agency” (Story, p. 159), political action on the part of actors arises primarily from their choices about what they consume and how they consume it. Consequently, production and consumption become the “two key moments in the circuit of culture” (du Gay, p. 5). The foregrounding of these two moments, which can be linked to cultural studies’ Marxist roots, has led Grossberg (1997) to assert that research using articulation theory often “fail[s] to rearticulate the forms of empowerment and survival with which people maintain some control over the construction of their own differences, their own lives, and their own possibilities, to the structures and tendential forces of the social formation” (pp. 12-13). This lapse weakens the ability of articulation theory to uncover the interplay between structure and agency, an uncovering that is necessary for understanding the possibilities by which actors can assert meaningful action in their lives.
Constructivist structuralism, the third approach to culture that I have discussed in this chapter, has also received a number of critiques (see Butler, 1999; Chazel, 1994; Eder, 1993; Sewell, 1992). Bourdieu’s emphasis on the “homogeneity of habitus” in his empirical explorations of different cultural fields (1977, 1984a, 1984b) is considered problematic because it seemingly fails to account for the ways in which change can and does occur. Bourdieu (1977), for example, writes, “One of the fundamental effects of the orchestration of habitus is the production of a commonsense world endowed with the objectivity secured by consensus on the meaning (sens) of practices and the world, in other words, the harmonization of agents’ experiences” (p. 80). Much like Foucault’s discursive formations, harmonization is structured through fields that provide the “rules” that make certain practices appropriate and others not, but does not point to the ways in which political action and change occur.

Butler, in her critique of constructivist structuralism, locates the source of this problem in the fact that the “…habitus presupposes the field as the conditions of its own possibility” and consequently, “the field…does not alter by virtue of the habitus, but the habitus always and only alters by virtue of the demands put upon it by the ‘objectivity’ of the field” (p. 117). Thus, according to Butler, the field determines habitus and even though habitus is the manifestation of the actions of actors, actors do not at the same time influence the field. The result of this imbalance is the reification of the objective/subjective dualism (p. 119). Even when habitus is conceptualized as the means by which strategies on the part of actors are made possible, a structuralist understanding of culture akin to the economic determinism is often advanced. For example, Bourdieu’s study of class (1984a) reifies a necessary relationship between economic class and socio-cultural dispositions, what Bourdieu calls “tastes.” Although Bourdieu states that strategic challenges to these dispositions and the structures that objectively shape them are possible, these challenges typically amount to nothing significant, merely a “misfire” (p. 5). In constructing this dialectical theory, Bourdieu makes little room for resistance on the part of actors or for understanding how resistance occurs. According to Sewell (1992), “what gets Bourdieu off
track is his unrealistically unified and totalized concept of habitus, which he conceptualizes as a vast series of strictly homologous structures encompassing all of social experience” (p. 16).

Although the critiques do not dismiss the importance of these theories of practice—in truth, these theories are, in the words of McAnulla in his critique of structuration theory, “a step forward” (p. 279)—we clearly need to engage in more critical examinations of these approaches and to suggest ways in which we can re-envision and add to them. In doing so, I believe we can further or own research practices, particularly in cases where the objects of study are both social practices and the macro- and micro-level contexts in which these practices occur.

Though research in the field of rhetoric has done much to evidence the ways in which structures shape discursive practices that function to reify and/or challenge the structure that shapes them, none of the studies that I reviewed, with the exception of Bazerman’s study of incandescence, has undertaken a lengthy examination of the broader discursive system in which practice occurs. I do not believe that this is some oversight by these researchers, not by any means, but instead might be a result of the familiar macro-level contexts surrounding their studies, all of which involve academia (Faber, 1998; Herndl, 1993; Herndl & Nahrwold, 2000; Johnson-Eilola, 1997; Sullivan & Porter, 1997), non-academic workplaces (Faber, 1997; Herndl, 1993; Slack et al., 1993), and/or relatively broad rhetorical discussions of genre (Geisler, 2003; Johnson-Eilola, 1997; Miller, 1994b). Because of shared disciplinary assumptions about the institutional workings of academia, for example, discussion of disciplinary research practices are somewhat unproblematically situated against structured, macro-level backgrounds. In a more in-depth study such as Bazerman’s, the context occupies as much of the discussion as the analysis of the discourse in a particular discursive formation, perhaps in part due to the “foreignness” of the space and time in which incandescence was developed and made meaningful.

To study the discursive encoding of software and the related practices of the software programming culture primarily responsible for this encoding, I have developed a framework that embraces the utility of theories of practice yet recognizes the limitations of these theories in resisting the dualism of objectivism and subjectivism. This framework, which I describe in
the following section, guides both my choice of texts from the software programming culture to analyze and the methods by which to analyze those texts.

**THE SPATIO-TEMPORAL FRAMEWORK**

What do I mean by “framework”? As Smith (1996) writes, “A framework points (more loosely than a formal model) to meaningful data in a situation” (p. 208). In my own development of an analytic framework for my study of the discursive coding of software by the software programming culture, I am motivated by Giddens’, cultural studies’, and Bourdieu’s respective theories of practice as well as critiques of those theories. Each of the theories of practice I discussed in the previous section agrees upon the importance of structure to agency and agency to structure. Giddens speaks of the recursive nature of structured systems and social practices in which systems are both the medium and outcome of social practices and practices, structured by the system, and also resources by which actors recreate those structures. In cultural studies, Slack and others emphasize the “non-necessary link” between meaning and practices and consequently call for studies of discourse to “map the context” in which “identities, practices, and effects generally, constitute the very context within which they are practices, identities or effects” (p. 125). And finally, Bourdieu emphasizes the ability of actors to make choices, though the “principles of these choices” are structured by the social system in which those choices occur (1990, p. 45).

Practices happen in particular moments of action undertaken by actors enmeshed in specific relationships of power. Because analyses of social practices are typically rooted in particular socio-historical moments, these analyses inherently involve issues of time and space. To understand the interplay of structure and agency in social practices that theories of practice elucidate—yet avoid the pitfalls inherent in these theories—I argue that we need to account for the role of space and time in this interplay. A framework built around the importance of space and time as well as structure and agency pointedly looks for the ways in which stability and change occur through shared cultural practices.

However, space and time, as analytic concepts in the study of social life, have received little attention in actual analyses of social practices compared to other cultural terms such as structure and agency. Commenting on this lapse, Grossberg (1996b) writes,
Often the most “obvious” feature of our experience—e.g. the distinction between time and space—are the least examined philosophically. I believe these fundamental structuring categories have major consequences for many of the theoretical problems of cultural theory—meaning and representation, subjectivity and agency, culture and society, identity and power. (p. 170)

This is not to suggest that space and time are never mentioned in theories of practice, but rather that these theories do not necessarily make space and time primary to the analysis of the interplay between structure and agency in social practices. Space and time are, to varying degrees, elements of interest in structuration theory, articulation theory, and constructivist structuralism for these elements are the means by which social practices are situated in context—the when and where of actors and events relative to each other and the context.

Giddens looks to context to understand how the “duality of structure” works in space and time at both the macro- and micro-levels. Similarly, cultural studies posits that locating social practices in space and time allows for “radical contextualism,” which, in turn, locates social practices as the bridge between the macro- and the micro. Finally, Bourdieu’s work in constructivist structuralism considers space as a site of relational positions that are historically structured through habits that can be strategically employed.1 However, given the critiques leveled against these theories of practice, I argue that what is needed in the

---

1 To expand on the use of space and time in theories of practice, Giddens, frustrated with the configuration of time and space as “mere environments of action” and with the failure of social scientists to think about the ways in which the modes of “time-space” constitute social systems, Giddens (1984) argues for a more concrete configuration of “the ‘situatedness’ of interaction in time and space” (p. 110) through attention to context. Similarly, cultural studies advocates “radical contextualism,” which describes as a researcher’s attention to “changing historical conditions, including the changing spatial and temporal complexity of the field of…culture” (Grossberg, 1992, p. 40). In seeking to uncover those factors that shape these practices, Grossberg states, “one must look…to the context, the dispersed but structured field of practices in which the specific articulation was accomplished and across which it is sustained over time and space” (p. 70). And, Bourdieu, in perhaps the most explicit use of space and time in his analytic concepts, argues that actors are able to strategize, a phenomena that occurs through the “reintroduc[ion] of time” (p. 9), as exemplified in the Kabylian ritual of gift exchange. In this ritual, a gift is given and a “counter-gift” given in return. How this counter-gift is given and received depends upon a strategy that has a “tempo” (p. 7). This tempo, “which consists of playing on the time…of the action” can configure the practice of giving a gift after receiving a gift as ungratefulness (the counter-gift is not given in reciprocation in an acceptable amount of time or at all), as obligation (the receiver of the gift is “obliged” to the giver until a comparable gift is given), or as friendship (comparable gifts are exchanged in an appropriately timely manner). As such, time plays an integral and generative role in social practices because “the same act…can have completely different meanings at different times, coming as it may at the right or wrong moment” (p. 6). Bourdieu also draws attention to the importance of space in constructivist structuralism through his concept of “field,” which, as I quoted earlier, is a “network, or a configuration, of objective relations between positions” (Bourdieu and Wacquant, 1992, p. 97). Positions are objectively mapped according to the structures of the field and the ways in which actors reproduce these structures in a constant negotiation of power. Within the “game” of culture, field is the space in which the game is played out by actors, who are invested to varying degrees, in the game itself. Thus, “a field is not simply a dead structure…but a space of play” (p. 19) in which actors reproduce the structure of the field through their actions which are shaped by the habitus.
methodology is a framework that emphasizes the dynamic nature of time and space in the study of social practices and the meaning constructed through discourse.

With this aim of a spatio-temporal framework, I have borrowed a three-pronged approach to time known as the “chordal triad,” developed through a review of research in the social sciences by Emirbayer and Mische (1998), sociologists who draw upon George Mead’s pragmatist work. In *The Philosophy of the Present*, Mead locates time in “emergent events,” which require a continual refocusing of past and future” and emphasizes the relational aspect of the social (as quoted in Emirbayer and Mische, p. 968). Action emerges, according to this pragmatist school of thought, “coterminously within contexts that are themselves ever changing and thus always subject to reevaluation and reconstruction on the part of the reflective intelligence” (p. 967-68). To this end, Emirbayer and Mische argue for the conceptualization of time as the temporal orientations of actors to the past, future, and present. With such a conceptualization, action can thus be understood dynamically and relationally.

Though Emirbayer and Mische do much to foreground the role of time in understanding social practices in context, they do little to explicitly explore the role of space in the relational nature of action. Consequently, I also draw from discussions of postmodern mapping of social positionings, a concept which I argue is enhanced by Emirbayer and Mische’s discussion of time. In doing so, my aim is to respect the differences between time and space while viewing them as intricately entwined in the interplay of structure and agency in social practices.

**Time as Temporal Orientations to the Past, Future, and Present**

In their own critiques of theories of practice, Emirbayer and Mische reject the mutual constitution of structure and agency that is common to these approaches because “the result has been a flat and impoverished conception [of agency] that…tends to remain so tightly bound to structure that one loses sight of the different ways in which agency actually shapes social action” (p. 963). To avoid a similar pitfall, Emirbayer and Mische emphasize the “double constitution of agency and structure” that shapes the relationships between actors, structure, and actions (p. 1004). This constitution embraces the possibility of actors to exert agency under the same structural influences in various indeterminate ways as a result of the
temporal orientations of actors to the past, future, and present. Thus, the deliberate actions of actors do not always already reify existing dominant structures; nor do their actions always already transform them. Instead, as Emirbayer and Mische argue,

all social action is a concrete synthesis, shaped and conditioned, on the one hand, by the temporal-relational contexts of action and, on the other, by the dynamic element of agency itself. The latter guarantees that empirical social action will never be completely determined or structured. On the other hand, there is no hypothetical moment in which agency actually gets “free” of structure; it is not, in other words, some pure Kantian transcendental will. (p. 1004)

In this way, the double constitution embraces the interplay of structure and agency without reifying the objectivist/subjectivist dualism that it seeks to challenge.

To describe how this double constitution actually plays out, Emirbayer and Mische conducted a literature review of social science research dealing with practice. From that review, they developed the “chordal triad,” which reflects Mead’s emphasis on the temporal and the relational in social life. Their analysis of practice-related research highlights the implicit thread of time that runs throughout studies of contextualized practices, though the past-orientation, with its attention to habitual practice and structure, tends to dominate such research. The chordal triad, on the other hand, aims to embrace the dynamics of time and thus is configured to emphasize

the temporally constructed engagement by actors of different structural environments—the temporal-relational contexts of action—which, through the interplay of habit, imagination, and judgment, both reproduces and transforms those structures in interactive response to the problems posed by changing historical situations. (p. 970)

In their review of social science research, Emirbayer and Mische locate the temporal orientation to the past, or “iteration,” in discussions of habit; the temporal orientation to the future, or “projectivity,” in discussions of imagination; and the temporal orientation to the present, or “practical evaluation,” in discussions of judgment. Though these varying temporal orientations are in a constant state of play with each other, research tends to emphasize the structuring effect of habit. To counter this trend, Emirbayer and Mische argue that reflective
actors are able to recognize the structural constraints on their actions by the social system in which they act as well as the possibilities that lie within these structures in a given context. Consequently, the mediation of the past and future influence the ways in which actors act in the present. As Emirbayer and Mische write,

> Actors are always living simultaneously in the past, future, and present, and adjusting the various temporalities of their empirical existence to one another…in more or less imaginative or reflective ways. They continuously engage patterns and repertoires from the past, project hypothetical pathways forward in time, and adjust their actions to the exigencies of emerging situations. Moreover, there are times and places when actors are more orientated toward the past, more directive toward the future, or more evaluative of the present; actors may switch between…their orientations toward action, thereby changing their degrees of flexible, inventive, and critical response toward structuring contexts. Such a perspective lays the basis for a richer and more dynamic understanding of the capacity that actors have to mediate the structuring contexts within which action unfolds. (p. 1012)

To detail the composition of Emirbayer and Mische’s temporal orientations as well as their relationships to one another, I next explain each element of the chordal triad. In doing so, I lay out the temporal feature of the spatio-temporal framework, a feature that encourages me to locate not only the stabilized cultural practices used by members of the software programming culture in their discursive coding of software but also the ways in which these habitualized practices are reflectively reconfigured and acted upon according to actors’ hopes and desire.

**Iteration as the Temporal Orientation to the Past**

The iterative element of the chordal triad emphasizes the temporal orientation of actors to the past and emphasizes the influence of structures that shape the everyday actions of actors in recurring situations according to the habits, routines, and dispositions of the social system. Emirbayer and Mische define iteration as “the selective reactivation by actors of past patterns of thought and action, as routinely incorporated in practical activity, thereby giving stability and order to social universes and helping to sustain identities, interactions, and institutions over time” (p. 971). Much like Bourdieu’s concept of habitus, which I
suggest is comprised of the iterative elements of a culture, for example, iteration acknowledges “typified recurrent patterns of experience,” which create “routines, dispositions, preconceptions, competences, schemas, patterns, typification, and traditions” (p. 971). These iterative elements create a recognizable stability in and uniqueness to a social system such as culture by continually reproducing a network of practices that remains constant because of actors’ “reactivation” of them on a continual basis with “a low level of conscious reflection” (p. 975).

Because Emirbayer and Mische are particularly interested in a theory of agency arising from temporal orientations, they seek to break from the apparent limitations that accompany such terminology, which often creates an overly structuralist account of action; they write, “most attempts to theorize the habitual dimension of action...focusing on recurring patterns of action themselves and thus upon structures, rather than upon the precise ways in which social actors relationally engage with those pre-existing patterns or schemas” (p. 975). Iteration, by emphasizing the reactivation of habitualized practices by actors through “tacit maneuvers,” reproduces a typified set of social practices, the set of which is specific each social system, without the structural determinism presented by other theorists (e.g., Bourdieu, 1977, 1984, 1990; Giddens, 1979, 1984) because reactivation through iteration represents only one aspect of time. With an emphasis on stabilization and continuity, analyses of habits, routines, dispositions, and typifications are thus only one means of identifying and describing social practices.

Projectivity as the Temporal Orientation to the Future

To complicate examinations of social practices that rely exclusively upon the habits, routines, dispositions, and typifications that are revealed through an emphasis on iteration, Emirbayer and Mische offer the concept of “projectivity.” Based on their review of the scant amount of research that emphasizes imagination, Emirbayer and Mische argue that much needed attention to the overlooked influence of the temporal orientation to future action, or projectivity, allows for an understanding of “the imaginative generation by actors of possible future trajectories of action, in which received structures of thought and action may be creatively reconfigured in relation to actors’ hopes, fears, and desires for the future” (p. 971). Projectivity emphasizes the creative abilities of actors to strategically imagine possible
trajectories of action that will help actors to achieve their goals. Emirbayer and Mische write, “Immersed in a temporal flow, [actors] move “beyond themselves” into the future and construct changing images of where they thing they are going, where they want to go, and how they can get there from where they are at present” (p. 984). Much of this imagination arises not only from their roles that are structured by the specific social systems of which they are apart—the software programming culture, for example—but also from the ideologies to which they ascribe, which are not necessarily specific to a particular social system.

Typically, the imagining of trajectories of possible action occurs through discursive practices such as “narrative constructions, symbolic recompositions, and hypothetical resolution” (p. 988). Such imaginings are crucial to “prophetic movements,” which reveal aspirations that may or may not support dominant ideologies (p. 922); “framing processes,” which “draw upon—and sometimes extend, rearrange, and transform—master frames”; and “institutional innovation,” which reflect the desires of actors to affect change within institutions. Projectivity then differs from iteration in that imagination inherently involves “reflectivity,” unlike the reactivation of habitualized practices through the temporal orientation of iteration. Instead, projectivity provides the means by which actors “give shape and direction to future possibilities,” (Emirbayer and Mische, 984) and thus entails reflection on the part of an actor.

**Practical Evaluation as the Orientation to the Present**

The third and final temporal orientation in the chordal triad is practical evaluation, which focuses on the temporal orientation of actors to the present. According to Emirbayer and Mische, practical evaluation addresses “the capacity of actors to make practical and normative judgments among alternative possible trajectories of action, in response to the emerging demands, dilemmas, and ambiguities of presently evolving situation” (p. 971). These trajectories of action are mediated by the habits, routines, and dispositions of iteration and imagination of projectivity. Because of this mediation, the ways in which actors might act are always structured to some degree by the iterative elements of a culture. But because of the capacity of actors to develop motives in accordance with their fears and hopes for the future, action remains ripe with possibility. Even in the most structured situations, actors,
inasmuch as they are aware of their own capacity to act in different ways in different contexts, retain the capacity to reify, resist, or transform. The possibility for resistance and transformation, in particular, is increased by the ability of actors to grasp the ambiguity inherent in situational contexts, and in doing so, deliberate upon, choose, and execute a course of action that is shaped by the structural core of the broader social system in which they act and yet, at the same time, reflects the aspirations of actors themselves. As Emirbayer and Mische note, “By increasing their capacity for practical evaluation, actors strengthen their ability to exercise agency in a mediating fashion, enabling them (at least potentially) to pursue their projects in ways that may challenge and transform the situational contexts of action themselves…” (p. 994). The ability of actors to make judgments and act deliberately necessitates an understanding of the context in which action occurs.

Configuring time as a complex relationship between the past as iteration, the future as projectivity, and the present as practical evaluation challenges the configuration of practices as always already determined or as always already indeterminate. Instead, practices are rooted in multi-temporality and are mediated by the structure shaped through the past and imagination of the future by actors. At the same time, the execution of practice in the present has the potential to shape the ways in which actors understand both the past and the future. In the case of my own study of the discursive coding of software by the software programming culture, this emphasis on multi-temporality provides me with a frame by which to understand action in this culture as the result of the mediation between the structured iterative practices that constitute the culture itself and the projective imagination of trajectories of possible action.

Though the chordal triad offers much by complicating the role of time in analyses of social practices, the “relational” aspect of the “temporal-relational contexts of action” (p. 1004) is not developed to the same extent that the temporal is. Or rather, the relational arises solely out of the temporal. After all, Emirbayer and Mische clearly state that their aim is “a more adequate theorization of the temporal nature of human experience” (p. 1012). But because space is integral to time and time to space, I contend that coupling a discussion of time with space increases the complexity of the “relational” that Emirbayer and Mische theorize through the “double constitution of agency and structure” (p. 1004) and the chordal
triad. As Crang and Thrift (2000), in their exploration of issues of space, suggest, “the fact remains that space without time is as improbably as time without space” (p. 1).
Consequently, I wish to add to Emirbayer and Mische’s work an explicit discussion of space as social positionings to illustrate the ways in which temporal orientations position actors in various sets of relations.

**Space as Social Positionings**

Discussions of space have become de rigueur in efforts to undermine the hold—some might even say the stranglehold—that history has had over accounts of the reality of social life. Recognizing the growing importance of space in the decentering of history, Foucault (1986) suggests, “the present epoch will perhaps be above all the epoch of space” (p. 22). Yet, this decentering has not been easily accomplished. Foucault’s own work has been critiqued for privileging time at the expense of space. In an interview with *Herodote*, its editors challenged Foucault about his emphasis on time at the expense of space: “You accord *de facto* privilege to the factor of time, at the cost of nebulous or nomadic spatial demarcation whose uncertainty is in contrast with your care in marking off sections of time, periods, and ages” (Gordon, 1980, p. 67). In response, Foucault argued that it is impossible for one person, as in the case of his study of prisons, to “cover the whole of this spatio-temporal field” (pp. 67-68). According to Foucault, to understand the discourse of punishment in the socio-historical context of his study in *Discipline and Punish* necessitated “looking at something that happened elsewhere in a more explicit form that antedated or served as a model for what took place in France” (pp. 67-68). Foucault’s configuration of space points to the relational aspect of space (i.e., the relationship between “a specifically French phenomena” and those similar, though in many ways different, processes that occurred “elsewhere”).

This notion of space, what I have been referring to as social positionings in order to emphasize the importance of space to social and the active processes, is rooted in the relational. Foucault’s discussion of space, in this instance, is clearly tied to the geographical. However, the importance of space is not limited to the physical. In the network society of which we are currently a part, space is much more than geography. Consequently, in this dissertation, I use the term ‘space’ not in geographical or physical terms but in terms of
social positionings, particularly ideological. Though conceptualizing space as such might lead to the charge of being nothing more than metaphorical, such metaphors are necessary because space is not something concrete that we can point to or step into. Space occurs through the ways in which actors live it. Space “…is not a neutral medium that stands outside the way it is conceived, we can trace, and dispute, various shifts in the organization of space alongside different forms of knowledge and social institutions” (Crang & Thrift, p.3).

Space-oriented research in professional communication, particularly research rooted in postmodern mapping, illustrates the ways in which space is manifest through social positionings that are not only relational but ideological (see Barton & Barton, 1993; Blair & Takayoshi, 1999; Cook, 2000; Grabill, 2001; Grabill & Simmons, 1998; Porter & Sullivan, 1993, 1997). Postmodern mapping is rooted in social constructionist approaches to visuals such as maps. For instance, Barton and Barton (1993), advocating “inclusionary design practice” (p. 77) in professional communication, argue that visual representations such as maps are not neutral markers of objective geographical locations. Instead, maps point to the ways in which social structures and ideologies are reproduced, consumed, included, and excluded, thereby positioning the objects mapped and not mapped, as well as the actors subject to these objects, in relationship to one another. These visual representations of geography are not objective visual representations but the means by which power and ideology are naturalized in specific socio-historical contexts. Consequently, postmodern mapping seeks to “denaturalize” such discourse to unveil visuals “as complicit with social control mechanism inextricably linked to power and authority” (p. 53). To do so necessitates the examination of the production of the discursive practice of map-making so as to understand the context that structures and is structured by the text, an objective that is important to both social and critical research.

But postmodern mapping is by no means limited to visuals. Other researchers in professional communication have used the method of critique behind postmodern mapping to examine the means by which social positionings are reproduced and resisted in English studies, technical communication, and workplace documents. For example, Sullivan and Porter (1993) locate the multiple social positionings of professional writing in relationship to other areas of concentration in English departments. In the traditional mapping of English,
literary studies is located at the center; consequently, all nonliterary texts, which includes composition and professional communication, are located outside the primary function of this academic field. However, a challenge to the centering of literature has occurred through rhetoric and compositions, which has come to be positioned as a “separate-but-equal discipline” because of its emphasis on rhetorical theory. In this mapping, professional writing can be positioned as a subdiscipline of rhetoric and composition or, like rhetoric and composition, separate but equal. In yet another mapping, Sullivan and Porter locate the social positioning of professional writing in relation to business and technical writing. Because professional writing is concerned with the “communication of professionals” and “communication in the professions,” professional writing is somewhat distinct from business and technical writing, both of which are tied “to a service-course identity” (p. 401). Yet, professional writing and communication does share some common ground with business and technical writing, although the exact connection remains complicated due to the unclear use of these terms in research, textbooks, and academic majors (p. 408). To manage this complicated set of relations, Sullivan and Porter call for a mapping of professional writing that differentiates it from other disciplines and subdisciplines within the field according to genre/place, perspective (idealistic or technical), and need (public rather than corporate) (pp. 412-414). By emphasizing the social positionings among areas of study in English departments, Sullivan and Porter emphasize the different ways in which professional writing can be located in relationship to other areas of study in a complex matrix of power involving the values and currency of the field.

In another postmodern mapping, Johnson-Eilola (1996) maps the shift from stories, particularly historical narratives, to space in postmodernism. Because of this spatial turn, Johnson-Eilola argues that technical communicators “must reposition ourselves as mapmakers rather than authors” (From Stories to Maps section, 3). This repositioning of technical communicators holds the potential for increasing the influence and status of the social position of technical communication. Transforming the work of the profession from helping consumers use products to “orchestrating a context, of arranging user, technology, and knowledge in particular, valued alignments” hinges on the abilities of our discipline’s professionals to construct maps that explicate the heterogeneity of communication practices
Rather than constructing communication as linear, context-free, and ideologically neutral, a postmodern mapping of discursive practices draws attention to the varied social positionings and sets of relations that are produced and reproduced through texts. Attention to space and the subsequent complicating of communicative processes offers the opportunity to improve the social position of those who produce these texts—technical communicators.

In her critical rhetorical analysis of the United States General Accounting Office’s (GAO) construction of a report addressing issues of sexual harassment at the U.S. service academies, Cook (2000) utilizes postmodern mapping theories to trace the structural constraints and rhetorical actions of the report’s writers. The governmental context of the GAO with its emphasis on objectivity, accountability, efficiency, and reliability necessitates that these writers emphasize the issue of data collection rather than the experiences of those subject to sexual harassment (pp. 69-70). This leads writers at the GAO in general and of the sexual harassment report in particular to “privileg[e] the interests of dominant power structures…at the expense of the sometimes inarticulate but, nevertheless, significant voices of individuals and their silenced expressions of suffering” (p. 55). Using postmodern mapping theories, Cook argues, can be beneficial to both writers of reports and researchers. Writers can use mapping to “conceive of their tasks” in context so as to go beyond unreflexive “reinforcement of the status quo” to a critical understanding and practice that recognizes the “social impact” of public policy documents and works towards improving the lives of those affected by issues under investigation, something that is often overlooked in reports created within the context of dominant power structures (p. 55). In the case of the sexual harassment report, “rather than recommending definitive actions to remedy, and possibly eliminate, acts of sexual harassment in the military academies, these writers deferred to the military and to its long-standing chain-of-command policies” (p. 56). Additionally, Cook suggests that postmodern mapping encourages a self-reflexive approach to research practices, an approach that necessitates an understanding of the “personal, institutional, and cultural contexts” that influence a researcher’s own social positioning. For instance, the specific mapping that Cook offers, one that centers on the cadets who had been harassed and who had been rendered relatively invisible in the report, is only one of many maps that can
be constructed from this rhetorical moment; she writes, “my concerns and biases, therefore, form a context for my own report...My map of the GAO report, consequently, reflects my own values” (p. 73). Postmodern mapping, Cook argues, offers methods of analysis that can benefit both workplace writers and workplace researchers through an emphasis on the socially constructed nature of rhetorical action in complex contexts that can be understood by mapping social positions.

Postmodern mapping emphasizes the importance of social positionings and the negotiation of power that occurs through these positions in context. For Sullivan and Porter, postmodern mapping afforded the means by which to understand the dynamic relationships between professional writing and other areas of study in English departments; for Johnson-Eilola, the means by which to re-envision the role of technical communicators so as to increase their status; and for Cook, the means by which both to critique contexts in which dominant power structures dictate discursive practices. Each imagines ways to offer real solutions to complex social problems and to situate the researcher in relationship to the research, thereby increasing reflexivity. Because mappings can vary, discussions of social positionings through the use of postmodern mapping are always partial and contingent upon the ways in which users, whether professional writers or researchers, configure the metaphorical, and in some cases actual, map. Thus, positivistic notions of truth and objectivity are antithetical to the spirit of postmodern mapping, which include the representation of the relationship between context and text with an emphasis on the power dynamics occurring through this relationship. However, explicit discussions of space often have little to do with explicit discussions of time, which is problematic if we agree that time is integral to space and space integral to time in understanding the interplay of structure and agency in contextualized social practices.

Consequently, by coupling the chordal triad’s emphasis on temporal orientations to the past, future, and present with explicit discussions of space as social positionings, I believe that I can offer a critical analysis of the discursive coding of software by the software programming culture. This spatio-temporal framework provides a frame by which to analyze the discourse of the culture that attends to the ways in which 1. the reactivation of typified iterative practices provide structure; 2. the projectivity of actors constructs a plurality of
ideologically encoded trajectories for future action; and 3. the mediation of these two temporal orientations shape the deliberative action of actors through practical evaluation. By also emphasizing the importance of space in these temporal orientations, I map explicitly the ways in which actors are socially positioned and organized in specific sets of relations. In doing so, I embrace the move by theories of practice to subvert an either objectivist or subjectivist reading of reality and seek to emphasize the interplay between structure and agency. Through the vantage offered by the spatio-temporal framework, I also hope to avoid the critiques of theories of practice by analyzing discourse in such a way that this interplay is always both stabilized and change inducing. By analyzing the discursive coding of software with this “double constitution of agency and structure” in mind, I hope to contribute to lay users’ digital literacy so that they can actively participate through their discourse in the meaning-making of software.

**METHODS AND CHOICE OF TEXTS**

I have argued that the foregrounding of space and time in the analysis of social practices encourages a recognition of the ways in which discourse creates both stability and change through the interplay of structure and agency. Guided by the spatio-temporal framework I built around the work of Emirbayer and Mische’s temporality and postmodern mapping’s social positionings, I describe in this section both my choice of texts to analyze and the methods by which I analyzed those texts.

**Iteration and Synthesis of Software Literature: Chapter Three**

To identify and describe the typified iterative practices that constitute the software programming culture would be a huge undertaking. Fortunately, researchers from other academic disciplines, commentators from the software programming culture itself, and government regulatory organizations have identified and discussed many of these recurring practices, though no one has until now grouped these together to create an overview of the everyday practices of the culture. For example, the respective work of Turkle (1984) on the computer science “nerds” at the Massachusetts Institute of Technology, Brooks (1975) on the importance of play to software programmers, and Levy (1994) on role of the “hack” in software programming allow me to talk about the practice of programming not as just a
functional exercise of intellectual work but as an act of technophilia, or love of technology. The U.S. government’s Office of Technology, U.S. Patent and Trademark Office, and National Commission on New Technological Uses of Copyright Works allow me to identify and discuss the regulatory practices used by the software programming culture to control software use. However, in this discussion of the iterative practices that provide stability and constituency to the software programming culture and its discourse, I chose not to include literature written with an overt material or ideological investment in software development. The one exception to this rule is the inclusion of Eric Raymond and his widely-recognized work on hackers. Otherwise, I excluded from Chapter Three the iterative practices of the software programming culture literature published by key individuals (e.g., Bill Gates, Richard Stallman, Linus Torvalds), software companies (e.g., Microsoft, Red Hat, IBM), and software organizations (e.g., the Free Software Foundation, the Open Source Initiative, the Business Software Alliance) within the software programming culture.

In the chapter, the discussions of software practices that I do synthesize are used to describe the habits, routines, dispositions, and typifications that together constitute the structured, stabilizing network of social practices that make the software programming culture distinct from other cultures. Included in this network are the practice of programming, instrumentalist talk about software, the discursive practice of evangelism, and institutionalized regulatory practices for controlling the use of software. I chose to include these practices for two reasons: 1. it is through these practices that ideology becomes most evident, as subsequent chapters in this dissertation illustrate; and 2. these practices appear to be the ones that most concern those who are responsible for the development and regulation of software. Not included in this discussion are practices such as usability testing and end-user documentation. I chose not to include practices such as these because in the four years spent researching the discourse of the software programming culture, few statements were made about them by members of this culture, though my own research in rhetoric, generally, and professional communication, specifically, made these practices particularly interesting to me.

Then, to illustrate the ways in which the “reactivation” (Emirbayer & Mische, p. 971) of these iterative practices by actors through their “tacit maneuvers” (p. 975) creates day-to-
day difference, I offer a brief overview of the reactivations involved in the development of Microsoft software, on the one hand, and GNU/Linux software, on the other. By comparing actual objects of software that were developed as a result of actors’ reactivation of the typified practices sanctioned and reproduced within the software programming culture, I illustrate how, in the words of sociologist Bell (1998, p. 54), how “difference through sameness” occurs in relatively unreflective ways. This illustration signals the different spatial positionings of actors who on account of the creation of different types of software, one an “established technology” (Christensen, 2003), the other a “disruptive technology.” The ideological significance of these positionings is explored in Chapters Four and Five.

**Projectivity and Ideological Analysis: Chapters Four and Five**

By moving to the temporal orientation of projectivity in the spatio-temporal framework for Chapters Four and Five, I begin an explicit analysis of what Grossberg called “the intervention of ideology into language” (1996a, p. 137) through the discursive coding of software by the software programming culture. To do so, I first analyze the narratives of spokespersons from different communities within this culture (Chapter Four). Second I illustrate the ways in which the trajectories constructed in these narratives are reified or challenged by the ideologically-disparate communities for which each of these spokespersons speaks respectively (Chapter Five). I do so by looking at the discourse of other key figures, organizations, and corporations that are apart of these respective communities. In mapping the ideologically-infused trajectories of action espoused by these different communities, I use Castells’ (1997) discussion of identity in the network society to explain and frame these always competing, though not always antithetical, discursive constructions.

In as much as narratives reveal the past in the present, narratives also configure the future. Emirbayer and Mische argue, “the construction of narratives…locate future possibilities in relation to more or less coherent causal and temporal sequences” (p. 989). Consequently, I examine the earliest narratives of spokespersons within the software programming culture. My choice to analyze ideologically the narratives of spokespersons arose from work of both Bourdieu and Castells. Their respective work on spokespersons is grounded in the recognition that collectives have leaders whom other actors recognize and endorse as spokesperson. Consequently, the discourse of spokespersons can be understood as
representative of the collective. Bourdieu compares this phenomenon to the Catholic religions’ ritual of transubstantiation, in which communion bread does not simply symbolize Christ’s body but is the body of Christ (1993, p. 247). In a similar fashion, a spokesperson does not simply symbolize the beliefs and desires of the group but is the group for all intents and purposes. The spokesperson “raises those whom he represents out of their existence as separate individuals, enabling them to act and speak through him as a single person” (Bourdieu, p. 248). Although this metonymic process wherein the part represents the whole may seem to impart all symbolic power to the spokesperson, the spokesperson is, in fact, dependent upon the power accumulated by the culture or community that s/he represents.

The authorized spokesperson is only able to use words to act on other agents and, through their action, on things, themselves, because his/her speech concentrates within it the accumulated symbolic capital of the group which has delegated him and of which he is the authorized speaker. (Bourdieu, p. 249) Thus, the power of a spokesperson is (re)produced with and through the communities for whom s/he speaks.

Much like Bourdieu, Castells (1997), in his study of social movements and identity in the Information Age, identifies a particular kind of spokesperson—the “prophet”—who represents the interests of communities and organizations that challenge and resist dominant interests (p. 361). Prophets act as symbolizers for collectives that challenge the apparatuses hegemony. These “symbolic personalities…give a face (or mask) to a symbolic insurgency, so that they speak on behalf of the insurgents,” (p. 361) an act of particular importance as subordinated or marginalized communities seldom have the access to capital, particularly economic capital, that dominant institutions do. Discourse, as a resource, comes to be an important means by which marginalized identities can accumulate power. Prophets, as those “who declare the path, affirm the values, and act as symbol senders, becoming a symbol in themselves, so that the message is inseparable from the messenger,” (p. 362) become in a way a capital in themselves.

Within the software programming culture, I identified two spokespersons—Bill Gates and Linus Torvalds—and one, in the words of Castells, “prophet”—Richard Stallman. Each is recognized as having made significant contributions to the software programming culture.
In addition, all three recognize themselves as the “face” of particular communities within the culture. Torvalds has referred to himself as the “chief nerd” in and “poster boy” of the Open Source community (Torvalds & Diamond, 2001, p. 188, 163). “Chief Crusader” is how Stallman (1999) describes his role in the Free Software community. And Gates (1994), the most well-known software-related personality in the world, who, obviously aware of his status, once commented during an interview, “if you want to just put ‘Microsoft spokesman’ next to all those comments, that would be fine…I know that people are more interested in human stories than they are in what technology can do for them” (Playboy, ¶12). Though other spokespersons could be identified with the respective communities that Gates, Stallman, and Torvalds respectively represent (e.g., Eric Raymond, Steve Ballmer, Bill Joy), no other actors are so publicly identified by the culture and by the communities than the three that I have chosen. Inasmuch as each represents a community with the software programming culture, each is also responsible in many ways for the development of that community. By examining the narratives of Gates, Stallman, and Torvalds, whom I refer to collectively as spokespersons, I rely upon Emirbayer and Mische’s assertion of the “importance of narrative construction in the development of collective projects of action” (p. 993).

In choosing which of the many narratives from these spokesperson to analyze, I focused my analysis on the earliest archived narratives in which Gates, Stallman, and Torvalds publicly announced, or if there was no announcement, mentioned for the first time, the seminal software development for which each came to be known. Doing so allowed me to trace the ways in which the ideological trajectory constructed in each of these narratives evolved by establishing stabilized meaning for software and the software programming culture or by instigating change in that meaning. In the case of Gates, the software was Altair BASIC, which he first publicly mentioned in “An Open Letter to Hobbyists,” first published on February 3, 1976 in Volume 2, Number 1 of the Homebrew Computer Club’s “Computer Notes,” a newsletter disseminated to computer hobbyists around the country. To ensure the accuracy of the narrative, I conducted my analysis from a scanned copy of “An Open Letter to Hobbyists,” which was made available along with the entire newsletter on the Internet by DigiBarn Computer Museum (http://www.digibarn.com/collections/newsletters/homebrew/V2_01/gatesletter.html). For Stallman, I chose, according to the specifications
that I had set, his initial announcement of the GNU Project: “new UNIX implementation.” This announcement was written on September 27, 1983 and posted via USENET, a distributed bulletin board system that predates the Internet, on the UNIX newsgroup discussion list, net.unix-wizards. To work from an authentic copy of Stallman’s post, I searched the archives of the USENET group net.unix-wizards and found the original positing of “new UNIX implementation” (http://comp.unix.wizards.net.unix-wizards/msg/4dadd63a976019d7?dmode=source). The earliest archived narrative in which Torvalds mentioned Linux, the operating system kernel for which he has since become famous, is “Free minix-like kernel sources for 386-AT.” Posted to the MINIX operating system listserve MINIX-L on October 5, 1991, this post marks the first time that Torvalds identified Linux. By searching the archives of MINIX-L, housed at the North Dakota University System website, I located Torvalds’ original posting (http://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind9110a&L=minix-l&T=0&P=8303).

To analyze the earliest archived narratives of these yet-to-be spokespersons, I used a method of ideological analysis used particularly well-suited to narratives. This method breaks down the narrative into elements of the story. In her discussion of narrative, Foss (1996) identifies and describes these elements as follows:

- Events—What is the plot?
- Causal relations—What are the cause/effect relationship of the events of the narrative?
- Narrator—Who is the narrator?
- Character—Who are the characters of the narrative and what are their traits?
- Audience—To whom is the narrative addressed?
- Theme—What is the significance of the narrative?

By analyzing each of these stories according to these elements of narrative, I sought to accomplish two tasks: 1. uncover the projectivity of these spokespersons, and hence, “the imaginative generation by [these] actors of possible future trajectories of action, in which received structures of thought and action may be creatively reconfigured in relation to actors’ hopes, fears, and desires for the future” (p. 971); and 2. map the spatial positionings
that set Gates, Stallman, and Torvalds, as well as their audiences, in specific sets of relations with one another.

To this end, I began my ideological analysis of Gates’ “An Open Letter to Hobbyists,” Stallman’s “new UNIX implementation,” and Torvalds’ “Free minix-like kernel sources for 386-AT” by situating each of these narratives in the evolving socio-historical context of the software programming culture (i.e., what was going on in software programming culture at the time in which the author wrote his letter or post?). This was facilitated by first analyzing the events constructed in each of these narratives and then situating these events in a broader socio-historical context of culture. This context-building allowed me to identity the causal relations not only between the events constructed in the narrative but also between those events and significant events occurring within the culture at the time.

Next, I analyzed the construction of the narrator by focusing on 1. the trajectory for action that the author constructed for the culture and its handling of software development and use; and 2. the social positioning constructed for the narrator as a result of this trajectory. The trajectory, in all three of the narratives, is constructed in response to a specific event that in the case of Gates and Stallman is made explicit and in the case of Torvalds is knowable only by situating his post in context. Each of these future spokespersons constructs the character of narrator in such a way that, through the ideological trajectory advocated in the respective story, the narrator is spatially positioned in various sets of relations of alignment and opposition with other characters in the narrative. I sought to uncover these sets of relations by comparing the construction of the course of action by the narrator with the course of action by other characters in the narrative. This allowed me to analyze the narrator and other characters in relationship to one another.

After analyzing the elements of events, causal relation, narrator, and character in the respective narratives of Gates, Stallman, and Torvalds, I then analyzed the ways in which audience was constructed as either spatially aligned with or against the narrator, thereby extending my analysis of the sets of relations constructed in the narrative. By examining the ways in which the narrator constructs his audience and the actions of the audience, I sought to understand the ways in which the narrator and his audience were constructed as sharing or
not sharing the same ideological trajectory. Additionally, I analyzed the ways in which social positionings of alignment or opposition participated in the construction not only of the trajectory advocated by the narrator but also the trajectory not advocated by the narrator. It is through an ideological analysis of these narratives specific to narrative that I was able to uncover a struggle for the discursive coding of software.

As a result of my ideological analysis of these narratives, I was then able to analyze the theme of the respective narrative of each future spokesperson. In analyzing the theme of each narrative, I drew conclusions about the broader significance of these narratives and how each of these narratives constructs a trajectory of action that is inherently ideological. It was a result of analyzing the competing themes of these narratives that I was able to frame ideologically my analysis of the narrative elements and the projective trajectories constructed through them.

But to illustrate the consistency of these discursive constructions from the time that the earliest archived narratives announcing or mentioning the seminal software projects for which these spokespersons would become known publicly to other members of the software programming culture, I conclude Chapter Four by offering examples of the ways in which these trajectories have consistently remained stable in the discourse of Gates over the last 30 years, Stallman over the last 23 years, and Torvalds over the last 15 years. Specifically, I examined texts constructed for the public; namely, speeches, court testimony, and published writings. Though I also sought examples that illustrated change in the narrative construction of these ideological trajectories, my research uncovered none. The projectivity of each of these spokespersons has remained stable and persistent.

Continuing my analysis of the projectivity constructed through the discourse of the software programming culture, I then analyze in Chapter Five the ways in which communities have developed that share and reify the trajectories constructed in the respective narratives of Gates, Stallman, and Torvalds. In Chapter Four, the analysis of these narratives and the sets of relations constructed by these stories illustrate the discursive struggle to articulate one ideological trajectory for the culture instead of others through the projectivity manifest in each of these texts. But as Chapter Five illustrates, this struggle not only contributes to the different social positionings of the characters of these narrative but also the
communities that have since developed around the projective imagination expressed in these narratives of those who were to become the spokespersons for respective communities within the software programming culture. To establish the significance of these communities beyond the software programming culture, I frame my discussion of the discourse of these communities in a modified version of Castells’ taxonomy of identity-building in the network society. In doing so, I illustrate the ways in which the projective trajectories constructed through the future-oriented discourse of the spokespersons and their respective communities are not ideologically specific to the culture itself, but instead are the result of identity formation that extends beyond the culture. In making this connection, I sought 1. to uncover the possibility of actors to reflectively imagine the ways in which the iterative cultural practices available to them not only structure their actions but also provide the means by which to enact their identities and corresponding ideologies in the everyday, and 2. to avoid making the narratives of Gates, Stallman, and Torvalds the singular cause for the identification of a specific ideological trajectory in the community that each spokesperson represents.

I located these communities as the proprietary software industry, the Free Software community, and the Open-Source community. Drawing from texts taken primarily from flagship organizations within these communities, I analyzed the ways in which these texts reify the projective trajectory constructed by the earliest archived narratives of each community’s respective spokesperson. Though I did include discourse from individual actors who hold positions of prominence in these communities (e.g., Steve Ballmer, the current CEO of Microsoft, and Eric Raymonds, a well-known figure of the Open Source community), my primary discussion of the discourse of these communities arose from the texts of the formal organizations that play such a significant role in these communities (e.g., corporations, advocacy groups, watchdog organization). In examining texts taken from the websites of these organizations, texts that included mission statements, FAQs, white papers, and legal documents such as annual reports, I also sought to include examples of discourse from within a specific community that challenged the narrative construction of the ideological trajectory reproduced by that community. In doing so, I was able to explain the sets of relations that include both alignment and stratification between the Free Software
community and the community that evolved out of it—the Open Source community. Except for a short period in which this change took place, my research did not turn up changes in the trajectories reproduced within the communities of the software programming culture, a fact that once again illustrates the relative stability of these competing discourses and the social positionings created through them.

Practical Evaluation and Genre Analysis: Chapter Six

In the final step of my analysis of the discursive coding of software by the software programming culture, I used the temporal feature of practical evaluation in the spatio-temporal framework. Practical evaluation, according to Emirbayer and Mische, is mediated by the temporal orientations to the past (iteration) and to the future (projectivity). By analyzing action that occurs in response to an emergent situation, I sought to examine the ways in which deliberative action of practical evaluation occurs through the mediation by actors of the structured practices of the software programming culture and the ideological trajectories constructed through actors’ and their communities projective imagination.

Specifically, I used the method of genre analysis to investigate the ways in which the shared, iterative regulatory practice of copyright, in the form of software licensing agreements, is “reconfigured” (Emirbayer & Mische) to align with the ideological trajectories forwarded by specific communities within the software programming culture. This process reifies the sets of relations constructed through narrative projectivity, and in the case of software licensing agreements, cements these relations in law. Theories of genre have evolved from decontextualized discussions of formal features to a situated activity of social action shaped by recurrence over time. As Miller explains in her seminal work on genre, “A genre is a rhetorical means for mediating private intentions and social exigence; it motivates by connecting the private with the public, the singular with the recurrent (Miller, 1984, p. 163). Though historically genre was used to classify texts into “different types” (Connors, p. 210) and those types into a taxonomy for modes of discourse (e.g. persuasion, argumentation, description, narration), the more recent work of genre theorists has expanded the concept of genre to include a dynamism that previous accounts of genre lacked (see Bazerman, 1994; Devitt, 1991; Miller, 1984, 1994b; Russell, 1997). As Russell states, the study of genre has moved beyond an emphasis on the formal features appearing across a
range of texts to analyzing genre as the “typified ways of purposefully interacting in and among activity system(s)” (p. 512). Genres are “tools,” and these tools “help mediate the actions of individuals with others in collectives…to create stabilizing for now structures of action and identity” (p. 514). Such an understanding of genre allows for the recognition of the ways in which genres “account for change as well as stability” (Russell, p. 516). Though the formal features of a genre may remain consistent, the “motives” behind the genre use may differ, and this can lead to instability and change. In short, genre is “both means and end, both resource and product” (Miller, 1994, p. 70).

Because copyright is the only regulatory practice used by all of the communities in the software programming culture, I chose to conduct a genre analysis of the different applications of software licensing agreements, the legal contract in which the licensor stipulates how software can be copied, modified, and distributed. The software licensing agreements that I chose to analyze are the agreements most closely associated with the proprietary software industry, the Free Software community, and the Open Source community, respectively. For the proprietary software industry, the software licensing agreement is Microsoft’s End-User Licensing Agreement (EULA). For the Free Software community, the software licensing agreement is the GNU General Public License (GPL). However, the Open Source community does not have one license that is commonly referred to as representative of the community. Instead, the Open Source community has a meta-license, the Open Source Definition (OSD). Unlike the EULA and GPL, the OSD does not regulate a particular piece of code, but rather the OSD stipulates what a software licensing agreement must do in order to be considered representative of the open-source approach to software development and use.

To understand these software licensing agreements and meta-license as the result of the deliberative action not just by their authors but by the communities that continue to use and license software in accordance with these agreements, I used Foss’s basic method of genre analysis, which includes “generic description,” “generic participation,” and “generic application.” First, I employed generic description to determine that a software licensing agreement genre existed. Having conducted a close reading in which I compared and contrasted the features of a range of software licensing agreements that included not only the
EULA, the GPL, and the OSD but also the Apache License, the GNU Lesser General Public License, and the Adobe End-User Licensing Agreement, I concluded that the shared inclusion of copying, distribution, modification, and warranty features in all (or almost all) these legal contracts pointed to typified conventions. Consequently, I concluded that these features constituted a genre that circulates both inside and outside of the software programming culture through the use of software by both members of the software programming culture and lay users.

Next, I used generic participation to confirm that the Microsoft’s EULA, the GPL, and the OSD included most (or almost all) of these conventions. Having confirmed that these licensing agreements and meta-license did, I next used generic analysis to evaluate them as “particular instances of rhetoric.” Unlike Foss, I was not interested in whether or not the EULA, the GPL, or OSD was “a good or poor exemplar of the genre” (p. 233). Through the guidance of the spatio-temporal framework, I was instead interested in the ways in which the differences in the application of the conventions of software licensing agreements could be understood as resulting from deliberative action mediated not only by copyright as an iterative practice with a specific generic structure but also by the projective imaginations that constructed specific ideological trajectories for the software programming culture. Additionally, in keeping with the importance of space to time, I choose to analyze the ways in which these applications reified or challenged the ideological social positionings constructed through the narratives of Gates, Stallman, Torvalds, and the communities for which they speak (Chapters Four and Five).

In the conclusion to my dissertation, I revisit the research questions that motivated my study of the discursive coding of software by the software programming culture. In particular, I make clear how my study offers opportunities for lay users to participate in the meaning-making of software.
CHAPTER THREE: THE REACTIVATION OF ITERATIVE SOCIAL PRACTICES

This chapter aims to answer the first of my research questions: What are the social practices by which ideologies circulate in the software programming culture? If, as Foucault and Hall suggest, social practice is the means by which meaning is made through discourse, then identifying and describing the typified practices of a particular culture allows us to begin to understand the cultural practices by which ideologies circulate discursively in the everyday activities of members of the culture. Because of the pervasiveness of software, these ideologies also circulate discursively in the everyday use of software by lay users. The set of practices that I identify and describe in this chapter, though not exhaustive of all the practices of the software programming, functions in a number of ways. First, these social practices differentiate the software programming culture from other cultures. Second, they provide stabilizing structure to the culture and its members because of the “reactivation” (Emirbayer & Mische) of these practices over time. And third, they reveal the ways in which the software programming culture is a uniquely structured social system, though at the same time influenced by institutionalized social practices, such as patent and copyright law, that affect other, differently structured cultures. In short, describing a set of taken-for-granted social practices that those in this culture reactivate on a day-to-day basis “with a low level of conscious reflection” (Emirbayer & Mische, p. 975) provides lay software users a general introduction to software programming as a culture and identifies possible avenues by which lay software users can participate—or be prevented from participating—in the practices by which the software programming culture discursively codes software.

To this end, I begin by identifying and describing the typified iterative practices that occupy much of the discourse within the software programming culture. In subsequent chapters, I explore how these practices are the means by which software is discursively coded. Together, these iterative practices constitute a network of social practices that distinguish the software programming culture from other social systems and provide structured stability to the culture and its members. These practices include the practice of programming, instrumentalist talk about software, the discursive practice of evangelism, and the regulatory practices used to control the use of software. But, in order to illustrate the ways in which these practices are reactivated “at a low level of conscious reflection”
(Emirbayer & Mische, p. 975), I examine the practices associated with the development of both established and disruptive software technology.

**THE PRACTICE OF PROGRAMMING AS KNOWLEDGE WORK**

Programming, the writing of coded instructions according to which all computerized technologies perform tasks, is perhaps the most easily identifiable practice of the software programming culture. As an iterative practice, programming provides structured stability to the culture—doing programming and talking about the doing of programming are essential everyday practices for many of its members. Additionally, programming is the knowledge work that, of all the practice discussed here, most clearly differentiates this culture from others and generates much of the culture’s intellectual, economic, social, and symbolic capital.

Knowledge work is qualitatively different from the typified work of the Industrial Age in that the practices associated with it result in ideas that allow others to do things rather than in tangible objects. With the move from industrialism to informationalism, knowledge work has become increasingly important to national and global economies. For instance, an Organisation for Economic Co-Operation and Development (OECD) report stated that knowledge work accounted for 7% of the U.S. gross domestic product in the year 2000 (OECD, 2003). Subsequently, the U.S. government’s Economics and Statistics Administration reported that software and software-related services generated $316.6 billion for the national economy (2004, p. 16). Knowledge-management expert Stewart (1997) describes this “New Economy” as follows:

> Information and knowledge are the thermonuclear competitive weapons of our time. Knowledge is more valuable and more powerful than natural resources, bit factories, or fat bankrolls…Wal-Mart, Microsoft, and Toyota didn’t become great companies because they were richer than Sears, IBM, and General Motors—on the contrary. But they had something far more valuable than physical or financial assets. They had intellectual capital. (p. xviii).

The knowledge work of programmers generates various kinds of capital for the software programming culture, whether intellectual, economic, social, and/or symbolic. Consequently,
programmers—those who do programming—occupy a highly-valued and integral position in
the culture and in society at large.

Modern programmers use programming languages to write the instructions for
computer hardware. A programming language is a text-based system that, because of its
similarities to natural language (i.e., a programming language follows rules of syntax and
semantics), is readable to anyone who has competency in the artificial language. Examples of
programming languages include Fortran, BASIC, C++, Java, and PHP. Programming
languages allow programmers to avoid the time-consuming work of translating instructions
into the machine-readable binary code (series of 1s and 0s) that computerized technologies
read to function. Additionally, programming languages allow programmers to read each
other’s work whenever the source code (the set of instructions written in a programming
language) is made available.

For example, when learning a new programming language, many programmers begin
with the simple program “Hello World!” (In fact, while taking a programming class in the 7th
grade, I wrote my first successful software instruction, which happened to be “Hello
World!”) When properly written, this command causes “Hello World” to appear on the
computer screen. In binary code, which is what the hardware reads, the command for “Hello
World!” might appear as follows:

```
11001001100001011010110000010101010101001001010010
1010100101100101010100101001010010010101010101
001101010101010010101010101010001010101010101
01010101010101010101010110010101011011011100
101011010110001001010100101011110010000100011101
1010001011010100100100100101010101010101001001111010
```

However, in the programming language C++, the program is written as follows:

```cpp
#include <iostream>

int main()
{
    std::cout << “Hello, world!\n”;
}
```
class HelloWorld! {
    public static void main (String args[]) {
        for (;;) {
            System.out.print("Hello World!");
        }
    }
}

From such elementary coding to advanced proficiency, programmers have created an essential component of the infrastructure that makes possible the networked society.

The birth of the modern software programming culture in many ways ushered in the New Economy of the Information Age. Castells (2000), a Professor of Sociology at the University of California, Berkeley, credits the revolution brought about by information technology to “the converging set of technologies in micro-electronics, computer (machines and software), telecommunications/broadcasting, and opto-electronics” (p. 29). Software programming, then, is involved in a complex matrix or what Castells calls a “network” of knowledge production made possible through “a process of technological transformation [that] expands exponentially because of its ability to create an interface between technological fields through common digital language in which information is generated, stored, retrieved, processed, and transmitted” (p. 29). For most people, the means by which to warehouse, access, and share information across networks are only available through the software programs that digital language makes possible. Although I do not suggest that technological transformation is made possible through software alone—certainly advancements in software are tied to advancements in hardware capabilities as my own brief history of the software programming culture evidences—software is a component integral to all of the technological fields that Castells mentions. The fact that “software has been the fastest-growing component of [information and communication technology] investment” (OECD, 2003) certainly illustrates the importance of programming in the knowledge work of the networked society of the Information Age.
Though the economic and intellectual capital that can be generated from programming is clear, as is the symbolic capital that both programmers and the software programming culture wield in our networked society (see Chapter One), social capital is also generated and embodied by what many other researchers of programming have described as the nerd/geek disposition and the programmer’s fascination with and love of technology. The nerd/geek disposition toward technophilia, or love of technology, is an often discussed both within and outside of the software programming culture. Most lay users of technology are familiar with the nerd/geek stereotype—a “too” thin or chubby male; he is socially awkward and a fan of Star Wars or Star Trek; he has a deep love of all things technology related—technophilia—and is devoid of any sexual prospects. Katz (1997), an observer of and author on software programming, defines “nerd” as a “term widely used to describe the sometimes awkward, technologically minded, gifted people who built the digital communication structures” (¶ 3). As such, none is nerdier than the computer programmer. Although Katz makes a distinction between nerds and geeks, the latter of which he describes as “less skilled in the mechanics of machinery… [and] more outward, political, and preoccupied with the applications of machinery and technology” (¶ 3), this distinction can be difficult to make as subsequent chapters of this dissertation make clear; consequently, I use “geek” and “nerd” interchangeably as is common in the discourse of the software programming culture. Though the stereotype has traditionally been a pathetic figure, the increasing pervasiveness of computer technologies in everyday life has raised the status of the nerd/geek whose role has become synonymous with power. PBS’s documentary “Triumph of the Nerds: The Rise of Accidental Empires” (1996) documents the role of the nerds in transforming society through technology, particularly the computer.

One of the first scholarly studies of the nerd phenomena and its relationship to software programming was an ethnography of undergraduates at the Massachusetts Institute of Technology (MIT) undertaken by Turkle (1984), a professor of Science, Technology, and Society at MIT. Rather than loathing the nerd/geek persona, Turkle observed many at MIT actually embraced nerdishness, as evidenced in MIT’s annual anti-beauty pageant: “The Ugliest Man on Campus.” Describing the event, Turkle writes, nerds “flaunt their pimples, their pasty complexions, their hobby knees, their thin, underdeveloped bodies” as they
compete for the title. And the royalty of the nerds at MIT are always the computer science majors, “the ‘out group,’ the ostracized of the ostracized” (p. 199). These students, who are not only seen but also see themselves as “the archetypical nerds…” (p. 200), divorce themselves in many ways from the body and its limitations to embrace the computer. Through this embrace, nerds become the “holders of esoteric knowledge, defenders of the purity of computation seen not as a means to an end but as an artist’s material whose internal aesthetic must be protected” (p. 207). Code creation is no mere task on the way to accomplishing some larger objective (e.g., doing complex mathematical equations) but a passion, a love, a source of fun. Not surprisingly, a search of Wired Magazine, a popular technology-related publication, results in no less than 200 articles including the words “software” and “nerd” and no less than 300 article including the words “software” and geek.” (Only 28 articles included “software” with both “nerd” and “geek.”)

But recognition of the technophilia-related traits of the “nerd” in software programming is by no means just an outsider’s observation. In The Mythical Man-Month, a seminal work on software project management first published in 1975, Brooks, a computer science professor who has made landmark contributions to software programming, addressed this question: “why is programming fun?” (p. 7). He attributes the fun of programming to the child-like enjoyment of “making things,” which evolves into the adult pleasure of the God-like creation of something distinct, new, and useful to others. Programming also fulfills the “nerdish” desire for constant learning, which manifests itself as a “fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle-cycles, playing out the consequences of principles built in from the beginning” (Brooks p. 7). Last, programming allows for an artistic creativity that Brooks eloquently describes:

There is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air from the air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures…Yet the program construct, unlike the poet’s words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The
magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be. (pp. 7-8)

The iterative practice of programming is not just a technical enterprise. For many programmers, code-writing is a means of creative expression, similar to that of a painter through canvas, brushes, and paints. Writing in programming languages also has the same issues of aesthetics that writing in a natural language does. And though many programming “authors” may share the same love of writing code, technophilia alone does not make every programmer into a programming equivalent of a Renoir, Picasso, or O’Keefe, or a Hemmingway, Wolfe, or Faulkner.

Brooks’ and Turkle’s identifications and discussions of the dispositions of programmers to technophilia as not just functional technique but as artistry and play is rooted in aesthetic traditions that pre-date the birth of the modern software programming culture. In his history of software programming, Levy (1994) describes the importance of the “hack,” which arose from the tradition of MIT students designing and implementing complicated pranks. In the 1950s, the MIT Tech Model Railroad Club (TMRC), a student organization dedicated to the hobbyist activity of creating elaborate miniature railroad displays, the hack was recognized as “a project undertaken or built not solely to fulfill some constructive goal, but with some wild pleasure taken in mere involvement…it would be understood that, to quality as a hack, the feat must be imbued with innovation, style, and technical virtuosity” (p. 23). For members of TMRC, many of whom were involved with the earliest incarnation of software programming, “technology was their playground.” Those that played on this playground came to be called “hackers.” a term that as the modern software programming culture evolved became a term of much contention, as subsequent chapters in this dissertation illustrate. Initially, the favorite toys of hackers were electronic gadgets such as model trains; as computer technology developed, software programming became the new and most fascinating toy of all.

More recently, Raymond’s *The New Hacker’s Dictionary* (1996), written to provide a “lexicon…to fledging hackers…” and preserve the values behind hacking (p. xviii), defined a hacker as “a person who enjoys exploring the details of programmable systems and how to
stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary (p. 233). Although “hacker” in lay society has been used to refer to someone who is illegally breaking into computer networks or creating Internet viruses, “hacker” is historically associated with a fascination with and love of code, though the term has become a significant point of contention within the software programming culture as subsequent chapters in this dissertation illustrate.²

Because the technophilia disposition of the nerd/geek usually revolves around the act of programming, the social relationships and positions among programmers are organized differently than in mainstream society. Although nerds/geek typically describes themselves as “loners” or “losers” (Turkle, p. 213), this description is primarily for the benefit of outsiders. The nerd life is, in fact, intricately bound to social customs and norms that are particular to the love of the technical endeavor of programming. Pavlicek (1999), a software programmer who also writes about geek life, comments, “Geeks are very social people...But their social interactions tend to follow the rules of geek culture much more than those of society at large” (p. 48). As a result, what is valued most is an individual’s ability at technical artistry rather than the number of users that one’s program might have or the amount of money the program makes. This artistry is what garners the reputation that programmers want and recognize as valuable to the geek in the software programming culture. As such, social capital, which is gained through reputation and ability in the practice of programming, is typically highly valued by programmers.

² To explain this contention briefly, “hacker” in lay society has typically been used to denote the exact opposite of what the terms means among nerds. Raymond, in The New Hacker’s Dictionary, states that coding to perform some illegal task is actually “cracking” not hacking. One who undertakes cracking is actually a “cracker”—“One who breaks security on a system.” The entry on cracker goes on to explain that the term was “coined ca. 1985 by hackers in defense against journalistic misuse of hacker” (p. 130). Because lay society primarily understands “hacker” to mean what is known in the software programming culture as “cracker,” some individuals, organizations, and communities in the culture itself have purposefully used hacker to signify those who were cracking because of the public’s misunderstanding of the term. For instance, Monte Enbysk, managing editor of Microsoft’s Small Business Center, posted on the company’s website an article about viruses in which defined “hackers” as “people who devise ways to break into networks” (7 things to know about virus writers). Having received complaints about his use of the term, Enbysk wrote in a follow-up column, “Indeed, all ‘hackers’ aren't criminals. Both good and bad share a common bond and form a highly caffeinated community where the lines get blurred” (Hacking into the mind of a hacker). To resolve the issue, Enbysk decided to “refer to the threatening ones as "computer criminals," "attackers" or "online thugs" — something other than "hacker" to avoid confusion or controversy.” Interestingly, however, the title of the article—Hacking into the Mind of a Hacker—reifies the public’s negative stereotype of hackers because the article’s topic is about the illegal activities of programmers and protecting the computer networks of companies from them.
But this is not to say that the social is valued to the exclusivity of economic capital in the software programming culture. It would be hard to argue such a thing when revenue stemming from the software and software-related services reached approximately $583 billion worldwide in 2005. With approximately $200 billion arising from the sales of pre-packaged software, software-related services comprise the majority of that revenue. Most evidently, we can see the embracing of the nerd/geek in computer-help businesses such as Geek Squad (www.geeksquad.com), Geeks on Call (www.geeksoncall.com), Nerds On Site (www.nerdsonsite.com), and NerdsUSA (www.nerdsusa.com), all of which offer phone support and house calls by nerds to help those struggling with hardware and software issues.

At the website of the Geek Squad, archetypal pictures of nerd employees in crisp white short-sleeved oxfords, black pants, black clip-on tie, black shoes, and, of course, white socks reify the physical image of those nerds described by Turkle at MIT’s “Ugliest Man on Campus” competition. And at the website of Geeks on Call, close-ups of faces wearing large black-rimmed glasses taped together at the nose offer another prototypical image of the nerd.

Though every computer programmer does not, of course, necessarily fit the image that we might conjure of the prototypical nerd/geek, a disposition is shared among those who lie at the heart of the software programming culture—programmers. With their love of technology, fascination with computers, and desire for capital that can and does come from their expertise, programmers, the primary, though not exclusive, members of the software programming culture are united through a shared technophilia indulged through the practice of programming.

**INSTRUMENTALIST TALK ABOUT CODING ISSUES**

Much of the everyday talk about software programming revolves around what I call coding issues. These coding issues, which function as habitualized discursive code topics, shape much of the instrumentalist talk of software, the second iterative practice that I have identified as an integral routine of many members of this culture. This instrumentalism emphasizes the effectiveness of code and is, quite possibly, the very kind of talk that makes it seem as if software is solely a technical issue best left to experts, rather than an ideological issue to which all lay users are subject. As with most instrumentalist discourse, the emphasis on efficacy can make such language appear ideologically-neutral; however, as much research
in professional communication has evidenced, technical talk is always ideologically-laden, which discursive analysis helps to uncover. Though I limit my discussion in this section to the identification and description of these coding issues, Chapter Four, with its emphasis on ideology, will implicitly make the connection between the discursive practice related to instrumentalist coding issues and ideology more salient.

These instrumentalist-oriented topics generally include, but are not limited to, issues of quality, which is often used as an umbrella term. Topics that affect the quality of software include, but are not limited to, security, flexibility, innovation, and transparency. In this section, I describe how these issues are used in relationship to the practice of programming and the object of software. Although these coding issues are intertwined with one another, I attempt to provide a basic understanding of each concept and how it relates to the other issues in the instrumentalist talk about software.

Quality

Issues of quality occupy much of the instrumentalist discourse of the software within the software programming culture. The quality of software, which is reflected in its performance, can be affected by, among other things, defects in software. These defects, commonly referred to as “bugs,” are estimated to have cost businesses and industries worldwide $175 billion (Innovation, 2004). Bugs appear to users in a number of ways, including slow or crashed operating systems, application shut-downs, and non-responsive or frozen programs. More serious defects have resulted in a number of critical errors over the years: the U.S. Treasury Department mailing 50,000 social security checks without any beneficiary names; the loss of the $125 million NASA Mars Climate Orbiter craft in space; the failure of a $1.2 billion U.S. military satellite launch; and false missile detection by the Soviet Union, which could have led to nuclear war in 1983. And, as examples in Chapter One illustrate, fatalities have occurred as a result of software bugs with estimates for the number of fatal occurrences at 428 between the years 1985 and 2003 (Newell, 2004).

Yet, bug-free software remains elusive because of the complexity involved in coding, even though the majority of time and money in software development is often spent on debugging. In A History of Modern Computing, Ceruzzi (1998) compares software to “an onion with many distinct layers of software over a hardware core” (p. 80). This complexity,
as well as the inevitable human factor, makes bug-free coding unlikely, though a recent study shows that post-release defects in every 1000 lines of codes has decreased from .46 in 1997 to .36 in 2001 (Rubin, Johnson, & Iventosch, 2002). With improvements in the quality of software, the reliability of software performance has increased substantially. Although software invariably has bugs, the key to quality coding is to make these defects as invisible to users as possible, and thus, render the program as intuitive and seamless as possible.

To celebrate its 2000th edition, ComputerWorld reflected on what it identified as the 10 biggest information technology happenings between 1986 and 2006. Included in those happenings is “Elusive Software Quality.” A search on google for information on “software quality” brings up no less than 6.99 million links, which range from guides to software quality, institutes for software quality, and software quality companies.

Security

To speak of quality and reliability is also to speak of software security. Hunker, dean of Carnegie Mellon’s school of public policy and management, notes, “The issues of ensuring software quality and security is one of the most important technical and public policy issues facing the nation and the world” (Innovation, 2004). The issue of software security speaks to the importance of, possibly even dependency on, code in everyday life. Without adequate software security, would-be information thieves can and have gained access to company databases, government facilities, and individuals’ personnel information—from bank statements to health histories. Additionally, software has to be secure enough so that viruses cannot easily infect computer systems, whether large or small. Yet, security is no easy task in light of a computer system’s onion-like design; “From a software quality and security point of view, an application inherits the problems of the external components on which it relies” (Whittaker & Thompson, p. 70). Thus, a program is often only as secure as the programs with which it communicates. Creation of firewalls, anti-viruses, and encryption programs are typical methods of increasing software security.

Flexibility

In addition to quality and security, flexibility is another key issue in software programming. Flexibility, also referred to as portability and interoperability, refers to the
ability of a program to run on multiple kinds of hardware and in conjunction with different
operating systems and applications. Because of the sheer number of possible hardware and
software configurations, programs must be adaptable and portable to different systems and
configurations. Moony (1992), a professor of computer science, commenting on this
diversity, writes

the continuing proliferation of both applications and computing environments creates
an urgent need to overcome the barriers of software portability. It is increasingly
likely that much of the software now in use or being developed will face the need to
be ported to new environments during its lifetime. (p. 53)

One way of ensuring a certain level of flexibility is the creation of standards that inform
software programming. Programming languages such as Java, C++, and XML have certainly
helped; however, flexibility continues to be an issue.

Another aspect of flexibility is the scalability of software. Scalability, which refers to
as the ability of a software application to adapt to a larger number of users than the
application was originally intended for, is also dependent on the power of the servers, which
allow multiple users to use the same program at the same time, and hardware abilities. Thus,
in the case of an e-commerce site, a programmer writes a particular piece of software to
handle the orders from customers. The scalability of the software is reflected in its flexibility
in handling a greater number of simultaneous users than the software might have originally
been intended to handle. With the continued growth of the Internet, this type of flexibility is a
key issue in creating software for the Web.

Innovation

Another key issue in instrumentalist discourse of software is that of innovation.
Software programming is intricately tied to hardware innovations, as I mentioned previously.
The faster the hardware can run, the more a software program can do at a faster speed. At
present, however, software developers have begun to question the possibility of truly
innovative discoveries in programming. As Love (2003), a software programmer, writes,
“Practically every piece of software that you currently use was once an innovation—Web
browsers, text editors, calculators, instant messengers, media players. Today, use of these
applications has become commonplace” (¶ 5). What has become more commonplace are
improvements to already existing software, typically referred to as new version releases. Yet, the quest to be innovative through software development remains a key coding issue.

**Transparency**

The last coding issue that relates to the evaluation of code is the level of transparency. The level of transparency determines whether or not other developers and users can actually view the source code of software (i.e., the programming language instructions in which the software is written). Typically, the transparency issue falls into two camps: those who do not make their source code transparent and thus black-box their software and those that do make their source code transparent. Black-boxed software “hides” the code by releasing the code in machine-readable language only. Releasing code in black-boxed form, which is typical of for-profit proprietary software, prevents programmers from modifying and/or distributing modified forms of the program. Transparent software, on the other hand, includes source code so that developers can copy, modify, and potentially distribute modified forms of the program. Free and open-source software is always released so that the code is transparent, thereby allowing anyone to modify the code to suit their particular needs. At present, the transparency issue is one of the most debated topics in the software culture, as I discuss in Chapters Four and Five.

**THE DISCURSIVE PRACTICE OF EVANGELISM**

In addition to the knowledge work of programming, another form of knowledge work functions as an iterative practice in the software programming culture—evangelism. Evangelism, which is sometimes also referred to as “advocacy” or “zealotry” by members of this culture, is a discursive practice by which the ideas of and about software programming are disseminated. Evangelism is not limited to the processes by which software is marketed, though that can be a purpose of evangelism; instead, evangelism is the talk about the benefits, problems, and larger significance of developing and using software in particular ways. Though the academic departments of computer science might seem like the primary outlet for such talk, evangelism is, in fact, distributed across the culture from individual programmers to global companies and can include topics ranging from issues of efficacy to ideology.
Evangelism in the software programming culture reflects what Chouliaraki and Fairclough (1999), in their re-imagination of critical discourse analysis in light of the Information Age, refer to as “linguistic reflexivity,” which they describe as follows:

Late modernity is characterized by an enhanced reflexivity (for example, in the construction of identities) which is in part linguistic reflexivity—awareness about language which is self-consciously applied in interventions to change social life (including one’s own identity). As commodities become increasingly cultural in nature they correspondingly become increasingly semiotic and linguistic, and language becomes commodified, subject to economically motivated processes of intervention and design (which involves linguistic reflexivity). (p. 83)

The linguistic reflexivity inherent in software evangelism is an integral aspect of the development of the new economy and the discursive coding of software. As the product of work for many becomes less and less oriented to tangible goods, the importance of the linguistic configurations of these “products” and their cultural significance become more apparent to those who participate in the discursive practice of evangelism. In the software programming culture, the “linguistic reflexivity” inherent in code-talk can address everything from nerd/geek technophilia and the best development processes to the clash between capitalism and the social good of society.

**REGULATORY PRACTICES**

To the network of iterative practices that I have thus far identified and described, I add the practices used to regulate the use of software: trade secrets, patents, and copyright. These institutionalized social practices of dominant society provide structured stability to the software programming culture through their continued use by actors. “Intellectual property” specialists (Neitzke, 1984; U.S. Office of Technology Assessment, 1990) have identified these practices as integral to determining what is meant by the lawful use of software. Additionally, these regulatory practices also make knowledge of the law, and hence, lawyers a vital component to the modern software programming culture.
Trade Secrets

Trade secrets were a particularly popular form of regulation for software when mainframe and mini-computers were the most common from of computer technology, although trade secret regulation continues to be used to a certain extent today (Office of Technology Assessment, p. 23). Because software was typically custom written for a particular computer or model of computer before the age of the PC, software developers commonly used trade secret law to protect code from “actual or threatened misappropriation” through use or disclosure by “improper means” such as “theft, bribery or espionage,” (National Conference of Commissioners on Uniform State Laws, 1985, section 2, 1). Although trade secret status is state-specific in the U.S. and country specific outside the United States when it does happen to exist in one form or another, many states have adopted the Uniform Trade Secrets Act. This Act defines a trade secret as

information, including a formula, pattern, compilation, program, device, method, technique, or process that

(i) derives independent economic value, actual or potential, from not being generally known to, and not being readily ascertainable by proper means by, other persons who can obtain economic value from its disclosure or use, and

(ii) is the subject of efforts that are reasonable under the circumstances to maintain its secrecy. (section 1)

Even though this Act creates a certain level of specificity as to what constitutes trade secret status, the concept, according to Frederic Neitzke (1984), remains “at best a nebulous concept which…is incapable of definition” (p. 26). Much of what constitutes a trade secret is left to “circumstantial factors,” which raises the question of how secret is secret (Alter, p. 91). Even using the federal protection associated with patents or copyrights for a program under trade secret status can nullify that status because federal regulations necessitate a date of publication, which, in turn, compromises the maintenance of secrecy.

In the age of the microcomputer, or PC, trade secrets have become the least likely form of institutionalized regulation used by software developers. The fact that software is now typically mass-marketed (though custom software projects are still common for business, industry, and government software-needs) raises potential problems regarding
secrecy, which is a key factor in this type of regulatory practice. The public-ness of today’s programs, even those that are released in black-boxed, machine-readable code only, make this type of regulation less than ideal for many software programmers and companies.

**Patents**

Patents, unlike trade secrets, are federal regulations, institutionalized through Article I, section 8 of the U.S. Constitution, which seeks “to promote the progress of science and useful arts, but securing for limited times to authors and inventors the exclusive right to their respective writings and discoveries.” Patents, as part of the iterative practice of code regulation, are often viewed as markers of innovation; the more patents that are issued, the more innovation that is taking place. Consequently, the number of patents awarded can be argued, in the words of the OECD, as “measure of innovation” (2003). The first software patent was awarded in 1968 to Martin Goetz, though the appropriateness of patents to software continues to be debated to this day (Lessig, 2002).

Patent awards are typically overseen by geographically- or country-specific government organizations such as the European Patent Office, which defines patentable software inventions as “an expression intended to cover claims which involve computers, computer networks or other conventional programmable apparatus whereby prima facie the novel features of the claimed invention are realised by means of a program or programs” (“Directives on the patentability of computer-implemented inventions,” Amendment 23, 2003). In the U.S. the patent process is overseen by the U.S. Patent and Trademark Office (USPTO). The process of applying for and receiving a patent entails establishing the usefulness, novelty, and non-obviousness of an idea, represented as an invention. If an invention meets these criteria, the patent, which protects the “*technological application* of an idea in a machine or process”, precludes others from applying or “practicing” the invention (Office of Technology Assessment, p. 8). Drawing from patent law, the USPTO states,

The right conferred by the patent grant is, in the language of the statute and of the grant itself, “the right to exclude others from making, using, offering for sale, or selling” the invention in the United States or “importing” the invention into the United States. As such, a patent is said to have “an exclusionary or negative right” (Poltorak & Lerner, p. 5). Just because a programmer is awarded a patent for a particular program does not mean, though,
that the programmer has a “positive right…to practice the patented invention.” Neitzke (1984) explains this distinction as follows:

…suppose that A has invented a novel software applications program, and that B has invented an improvement which used the basic program of A in a nonobvious manner. B could obtain a patent on his improvement, but could not practice it without infringing A’s patent. A could not practice B’s improvement either, without infringing B’s patent. (p. 39)

What typically happen in an event such as this is an agreement between the two patent holders so that both A and B can practice the application program with the improvement.

Patents fall into three categories: plant, design, and utility. Within the software programming culture, utility patents are the most common. A utility patent “cover[s] a device or an article, a composition of matter, a method or a process of doing or making something, or, less commonly, a new application for an existing device or material, or a product…made by a particular new process” (Poltorak & Lerner, p. 2). According to the most recent data available through the USPTO, the number of utility patents classified as Class 717, one of just many classes involving software-related patents, software development, installation, and management alone increased from 102 in 1997 to 357 in 2001 with IBM, Sun Microsystems, Hewlett-Packard, and Microsoft receiving 43% and 58% of the patents, respectively. Although comprehensive data regarding the number of patents awarded for software and software-related services is difficult to pinpoint because of cross-referencing practices by the USPTO, the OECD estimates that the number of patents awarded “grew from fewer than 5,000 per year in 1990 to approximately 20,000 in 2000, or approximately 15% of all US patents granted in that year” (2004, p. 24).

Copyright

Copyright is the most common form of institutionalized regulation in the software programming culture. Software copyrights speak to the rights and limitations involving the copying, distribution, and modification of programs. Like patents, copyright in the U.S. originates from Article I, section 8 of the U.S. Constitution. Internationally, copyright is protected by the Berne Convention. Unlike patents, copyright prevents only the copying of
the “expression” of an invention, not the practice of it. As Poltorak and Lerner (2002) explain,

A work created by another, without copying, is not an infringement, no matter how similar it may be to a copyrighted work. Moreover, copyright protects only the expression of an idea, not the idea being expressed. Thus, information or data included in copyrighted work is not protected against appropriation and use by others, although copying of the presentation and arrangement is barred. (p. 30)

Copyright law insists that authors and inventors have an automatic right to the protection of the “expression” of their work from being copied, distributed, and modified for derivative works in ways that infringe on the author’s expression of an idea. Consequently, the first copyright of software occurred with the first coding of software.

Yet, the applicability of copyright to software programs was not always clear. In 1978, at the behest of the U.S. Congress, the National Commission on New Technological Uses of Copyright Works (CONTU) issued a report that clarified the connection between new technologies such as software and copyright law, which had been newly revised through the 1976 Copyright Act. Before the birth of the microcomputer, the connection between software and copyright mattered little because most software programs were hardware-specific. Consequently, software had little need for protection as it was typically regulated by the protection associated with the specific machine or machine model on which the software ran.

According to the 1978 CONTU report, the introduction and proliferation of the microcomputer as a general-purpose machine allowed for a re-envisioning of software design and production. Cognizant that independent software producers such as the then named Micro-soft were cropping up and that computer and software production were no longer part of a mutual production process, CONTU determined that

if the cost of duplicating information [such as a software program] is small, then it is easy for a less than scrupulous person to duplicate it. This means that legal as well as physical protection for the information is a necessary incentive if such information is to be created and disseminated. (p. 24)
Thus, software is an “expression” and consequently, deserves protection under Article I, section 8 of the U.S. Constitution, specially, and copyright law, in general. However, the conception of programmers as authors or inventors was an additional issue that CONTU had to address because authors communicate stories or meanings to others and inventors create an object that can be used. What is the meaning embedded in machine language and the usefulness of the practices it offers? The communicative aspect of code and its uses are after all understandable only by machines.

In its interpretation of writing according to the Constitution, the CONTU report compared software code to a “series of meaningless words coined by a copyright claimant for use as a code for sending cable,” which Judge Learned Hand had found applicable in a previous examination of copyright and a new technology.

If…models or paints are “writings,” I can see no reason why the [meaningless code for sending cables]…should not be such because they communicate nothing. They may have their uses for all that, aesthetic or practical, and they may be the production of high ingenuity, or even genius…[O]ur Constitution [does not]…embalm inflexibly the habits of 1789 [when the Constitution was established]…Its grants of power to Congress compromise, not only what was then known, but what the ingenuity of men should devise thereafter. (pp. 35-36)

Working from Judge Hand’s interpretation, CONTU affirmed the fact that the programmer enjoys the same protection of his or her work as any other artist. Consequently, traditional notions of “meaning” and “aesthetics” do not determine what is copyrightable.

In 1980, Congress, acting on recommendations made by CONTU’s report, made amendments to the 1976 Copyright Act and explicitly extended copyright protection to software programs, which were defined as a “set of statements or instructions to be used directly or indirectly in a computer to bring about a certain result.” The extension of copyright to include software reads as follows:

Copyright protection subsists…in original works of authorship fixed in any tangible medium of expression, now known or later developed, from which they can be perceived, reproduced, or otherwise communicated, either directly or with the aid of a machine or device. (p. 12)
Through this extension, the right of software programmers to copyright protection, which due to the Sony Bono Copyright Extension Act extends protection for up to 95 years, was made explicit. The number of copyrights protecting software programs is unknowable as a programmer or company does not need to apply for or file the copyright. However, programmers can register their copyrights through the U.S. Copyright Office.

Trade-secret status, patents, and copyright are the typified means of code regulation in the software programming culture. In addition to these means of regulation, the practice of programming, the instrumentalist talk about software, and the discursive practice of evangelism constitute a network of practices that together create stability for the software programming culture and its members. Because of the iterative nature of these practices, actors within this culture reproduce the structure that these practices afford through their everyday routines, habits, and dispositions. However, the structuring effect of these practices does not necessarily mean that actors enact these iterative practices in exactly the same way. The difference that can and does arise within a culture is also a result of these shared practices. Even with little reflection, actors create difference while they reaffirm the shared iterative practices through their “reactivation” of these practices “tacit maneuvers” (Emirbayer & Mische, p. 975). I argue that the differences that occur through the reactivation of these iterative practices can be understood as a result of actors’ use of these iterative practices as a resource that, rather than prescribing how these practice work in the everyday, give members of the software programming culture the means by which to create difference even at a functional level. To illustrate the functional difference that exists within the software programming culture, I give a brief overview of the practices associated with two different kinds of software technology.

**THE CREATION OF FUNCTIONAL DIFFERENCES IN SOFTWARE**

The creation of functional differences in types of software is a result of the habitualized reactivation of the network of iterative cultural practices that I discussed in the previous section. To illustrate the difference that occurs through reactivation, I offer a brief example of two types of software that are a result of the reactivation of the habitualized routines of the software programming culture. One I classify as what Christensen (2003) calls an “established technology,” meaning that the reactivation of certain iterative practices has
created stability in the software programming culture for relatively long period of time. The other I classify as what Christensen calls a “disruptive technology,” meaning that the reactivations of certain iterative practices differ from the reactivations that led to the creation of the established technology. I offer this comparison between established and disruptive technology in the software programming culture to show the difference that can occur through reactivation of the network of iterative practices, a reactivation that occurs with little to no reflectivity.

**Established Software Technology**

The reactivation of the routinized practices of the software programming culture in specific ways has led to the development of software as an “established technology.” As Christensen, an associate professor of business administration at the Harvard Business School, explains, an established technology dominates the market. He lists traditional classroom instruction, laptop computers, and physical retail stores as examples of established technologies. Within the software programming culture, established technology includes Microsoft’s Windows operating system and the word processing application Word, Sun’s Fire server, and the web browsers Internet Explorer and Yahoo. Though these programs and applications range in both age and purpose, each of these examples is the result of the reactivation of the certain iterative practices of the software programming culture in particular, though not necessarily very reflective ways. Figure 3.1 illustrates this process of the reactivation of the shared network of cultural practices that constitute the software programming culture and the effect of this process, the development of established software technology that dominates software use.

Figure 3.1: Established Software Technology through Reactivation
The shared cultural practices (left box) constitute the culture and provide it with structured stability. The reactivation of these practices (center box) locates the shared iterative practices of the culture as a resource for the culture’s members. The result of these reactivations, in the case of the example at hand, is established software technology (right circle).

The practice of programming, for example, is conducted within corporations that have a traditional vertical hierarchy according to a linear development process. Bank (2001), a journalist who was given unparalleled access to the internal workings of Microsoft, describes this hierarchy as

- a shifting collection of teams, grouped into products, managed in divisions that correspond to Microsoft’s major product lines…Reorganizations are constant, as market shifts and new initiatives require additional resources. Vice Presidents, like coaches during an expansion draft in professional sports, protect their starts and raid other teams for specialized success. (p. 49)

To create software, Microsoft follows a linear development process in its programming practice. Barr (2000), who spent 10 years as a programmer at Microsoft, describes the coding process as a hierarchy involving project managers (PMs), developers, and testers (p. 47). PMs “write the specification…for a project, and then do whatever is necessary to ensure that the product winds up matching the spec.” Developers are responsible for the actual code writing, which is “where the real artistry occurs” (p. 49). After writing a patch of code to be included in the operating system, application, or software tool, developers “check in” the code into a “build” that houses all the code that is being developed for a specific program. Farther down the hierarchy of the “Microsoft geek culture” are testers, “the third class of technical people” (p. 154). The process of testing, in a linear development process, involves mundane code debugging and writing applications to test the software being developed (p. 50). In describing the ways in which this process typically works, Barr writes,

A large project may have hundreds of developers working on it, but the tasks will have been divided and subdivided until you have a particular piece of functionality that is owned either entirely by you, or by you and a few other people that you work closely with. (p. 41)
This is the “tacit maneuver” (Emirbayer & Mische, p. 975) by which the practice of programming is typically reactivated by vertically-organized corporations. The end result of this practice has established software technology that is black-boxed, meaning that the practice of programming does not include providing source code to those outside of the corporation or sometimes even within the corporation.

In another example of the way in which an iterative practice of the software programming culture is reactivated in ways specific to established technologies is the discursive practice of evangelism. Using Microsoft as an example again, the practice of evangelism has historically been reactivated through designated channels within the corporate structure, specifically the Developer and Platform Evangelism division. In his testimony in State of New York vs. Microsoft, Gates describes the practice of evangelism within this division as follows:

> We call the employees who spread the word of the platform benefits of MS-DOS “evangelists,” reflecting our passion and commitment with which they went about their jobs…Our Developer and Platform Evangelism Division, some 2,500 employees strong is dedicated to this work. Hundreds of evangelists are stationed around the world to assist developers in the building of products that interoperate with Windows. (¶ 29)

Just as in Microsoft’s reactivation of the practice of programming, the practice of evangelism is organized according to a corporate structure. The aim of this evangelization, which includes instrumentalist talk about the quality of Microsoft products, is the continued development of Microsoft-compatible products that reaffirm the continued dominance of Microsoft software, which is an example of an established software technology.

To control the use of its black-boxed software, Microsoft uses whichever regulatory practice it deems best suited to the job. Consequently, Microsoft has employed trade secrets, patents, and copyright to designate the legal use of its black-boxed software and to ensure that the source code for the majority of its software, including Word, Windows, and Internet Explorer does not become transparent.

Such reactivations of these iterative practices by Microsoft, as well as other corporations, are associated with a particular kind of software technology—established
software technology—that has historically dominated not only the software programming culture but also society at large. However, these reactivations are not the only way in which the habitualized cultural practices have been enacted through the day-to-day activities of members of this culture. In fact, these different reactivations have led to the creation of software that threatens to disrupt established software technology.

**Disruptive Software Technology**

Though the shared cultural and iterative practices of the software programming culture provide structured stability to the culture and its members, these practices also provide the means by which to enact difference, even though actors in their process of reactivating these practices may do so for functional, rather than ideological, reasons. The outcome of these reactivations that differ from those discussed in association with established software technology is the creation of disruptive software technology.

According to Christensen, established technologies are challenged, sometimes even replaced, by disruptive technologies that develop through innovation. Citing distance learning, handheld mobile devices, and e-commerce sites as examples of disruptive technologies, Christensen contends that disruptive technologies do not necessarily succeed because they are better than the established technologies they challenge. Sometimes the disruptive technologies have “worse product performance” (p. viii). But the danger of disruptive technologies is that they can, regardless of performance or quality, topple the dominance of established technologies:

Disruptive technologies bring to a market a very different value proposition than had been available previously. Generally, disruptive technologies under perform established products in mainstream markets…Products based on disruptive technologies are typically cheaper, simpler, smaller, and frequently, more convenient to use. (p. xviii)

Though established software technology is far from the verge of disappearing, the reactivation of the shared iterative practices of the software programming culture has led to the creation of disruptive software technology in the form of transparent source code software. Examples of disruptive software technology include GNU/Linux operating system,
the word processing application OpenOffice, the Apache server, and the Mozilla and Firefox web browsers.

In the development of transparent source code, the programming process, as illustrated in Figure 3.2, is parallel and open rather than linear and controlled. The shared cultural practices (center box) constitute the culture and provide it with structured stability. The reactivation of these practices (the box to the left of center circle) represents the reactivation of these practices and thus locates the shared iterative practices of the culture as a resource for the culture’s members. The result of these reactivations, in the case of the example at hand, is disruptive software technology.

As in the case with transparent software development for the GNU/Linux operating system, the transparent source code of this software allows anyone to see the source code and anyone with the necessary skills to develop code and test software. Programmers are not neatly divided into developers and testers, for instance. Instead, it is up to individual programmers and informal programming groups, and organizations, as well as corporations affiliated with the parallel development of transparent source-code software, to choose what to develop and test. In some cases, it is even permissible to modify the source code of someone else and redistribute it, a process that is not within the realm of possibility in black-boxed software because the source code is not readable.

Likewise, rather than having a specific division primarily responsible for the practice of evangelism, as in the case of Microsoft’s Developer and Platform Evangelism division, evangelism is dependent upon many of the actors, groups, and organizations involved. Though this does not mean that corporations are not involved with the development and evangelism of transparent source-code software (IBM after all is a prominent source of such
evangelism), the process is more diffused, just as the practice of programming is. And to control the use of transparent source-code software, only one of the regulatory practices is customarily used—copyright.

Identifying and describing the shared iterative practices of the software programming culture and the ways in which these practices are reactivated differently allows lay users to consider four interconnected aspects of the culture primarily responsible for the creation and regulation of code: 1. the practices by which the discursive coding of software typically occurs; 2. the “rules of formation” (Foucault) that give regularity, stability, and structure to the verbal network through which software is discursively coded; 3. a broad understanding of how the software programming culture works on a everyday basis that, with complication through further analysis in the subsequent chapters, provides a broad context for understanding the culture and its discourse; and 4. the culturally-sanctioned means by which power typically circulates within and through the software programming culture. Awareness of these iterative practices and their effect in the constitution of the discourse of software is a first step through which lay users can come to know the design of the software programming culture, and, in coming to know this culture, to identify those already naturalized practices that provide possible avenues by which they may or may not participate in the discursive coding of software. (I would hardly argue that every lay user should run out and become a programmer, for example, though the practice of evangelism possibly offers a means by which to participate in the meaning-making of software.) In doing so, lay users make a significant contribution to their own digital literacy and increase the possibility of the effective exercise of power through discourse.
CHAPTER FOUR: AN IDEOLOGICAL ANALYSIS OF THE ENCODING OF SOFTWARE

In both this chapter and the next, I answer the following research question: What are the political ideologies encoded in software? In the previous chapter, I identified and described a set of typified cultural practices that members of the software programming culture reactivate through their everyday routines and habits. I now turn to the study of the political ideologies that, through the discursive coding of software, give wider significance to both established and disruptive software, not only for those in the software programming culture and but also for lay users of software. My emphasis on the ideological, and hence political, encoding of software and software programming opens up the opportunity to analyze the ways in which individual actors and the communities to which they belong use structured iterative practices strategically as resources for enacting specific ideologies. The enactment of ideologies has consequences not only within the culture of software programming but also across society as a whole. The connection between ideology and the discursive coding of software begins to take shape in this chapter and continues in the next.

In this chapter, I identify the ideological encoding of software, its discursive coding, by analyzing the narratives of spokespersons from different communities within the software programming culture and then examining the ways in which the trajectories constructed by these narratives have remained stable for each of the narrators. And, because issues of ideology involve collectives rather than individuals (Hall, 1996), I examine in the chapter that follows how communities that both sanction and reproduce the discursive coding of software have emerged and coalesce around the narratives of these communities’ spokespersons.

To this end, I begin with an analysis of the earliest archived narratives by those who later became key figures in the discursive coding of software: Bill Gates, co-founder of the software giant Microsoft; Richard Stallman, founder of the GNU Project; and Linus Torvalds, the initial developer of the Linux kernel. Through their narrative constructions, each of these figures struggles to fix—in both the sense of correcting and stabilizing—the symbolic meaning and hence ideological significance of software. At the same time, these members of the software programming culture, each of whom was to become a spokesperson
for a community within that culture, imagine through their projectivity, or temporal-orientation to the future (Emirbayer and Mische, 1998), ideological trajectories of action intended to induce change in stabilized meanings of software development and use. The discursive constructions of these competing ideological trajectories carve out for each community possible and desirable paths for future action that also reify the shared iterative practices that structurally constitute the software programming culture and differentiate it from other cultures (see Chapter Three). In truth, the shared cultural practices make the narratives possible to a great extent. The construction of the narratives themselves should be understood as a practice of evangelism on the part of their authors. Through these narratives Gates, Stallman, and Torvalds spatially position themselves within a shared social system of the software programming culture but in competing sets of relations, what Foucault calls the “multiplicity of force relations” (1979, p. 92). These relations configure the characters of the narrative, as well as the narratives themselves in various positions of alignment and stratification through each author’s projectivity.

**STORIES OF SOFTWARE**

Storytelling is integral to identity-building and, as such, to the relational circulation of power (Ang, 2000; Castells, 1997; Faber, 2002; Hall, 1990, 1996; Kiecolt, 2000; Smart, 1999). Using ideological analysis allows me to emphasize the relationship of stories to the past and present in the reproduction of routines. However, Emirbayer and Mische argue in their review of the temporal dimensions of agency in research in the social sciences, the foregrounding of the future through attention to the projectivity of actors in discursive constructions such as narrative can help to reveal “the imaginative generation by actors of possible future trajectories of action, in which received structures of thought and action may be creatively reconfigured in relation to actors’ hopes, fears, and desires for the future” (p. 971). Accordingly, by drawing attention to projectivity and later its role in the mediation of deliberative action, we can better understand the ways in which actors “reflectivity” respond to uncertain or problematic situations in ways that support their ideological beliefs. Using ideological analysis to examine narratives, I identify the trajectories for possible future action, trajectories according to which actors might reconfigure the iterative practices of the software programming culture.
In following section, I analyze the narratives of the three software programmers who were to become spokespersons within and for the software programming culture. These spokespersons whose “speech concentrates within it the accumulated symbolic capital of the group which has delegated him and of which he is the authorized speaker” (Bourdieu, 1993, p. 249) include Gates, Stallman, and Torvalds, all of whom recognize their positions as spokesperson, as I discussed in Chapter Two. Specifically, I argue that Gates, Stallman, and Torvalds, through the projectivity manifest in their respective narratives that arise out of the shared cultural practices can be understood as examples of evangelism. Each constructs a competing ideological trajectory for the software programming culture and its members and each seeks to discursively code software differently, according to his values and beliefs. These always competing, though not always antithetical, trajectories are imagined in response to each narrator’s identification of a perceived problem in an emergent situation within the software programming culture. The narrative constructions of these problems and their corresponding solutions are born of beliefs about what the development and use of code should signify beyond its functional purpose of providing instructions to a piece of computerized hardware.

These narratives of software represent attempts to stabilize a particular meaning of software by challenging and or changing another meaning, to define the culture of software programming, and to advocate for different versions of the hopes and desires that the culture should fulfill. Significantly, in the respective stories constructed by Gates, Stallman, and Torvalds, the representation of software is intricately tied to issues of democratic freedom, though the “shape” that freedom takes is different for each. But these competing articulations of trajectories for achieving freedom, which also include warnings about the perils of not following these maps of possible action, do not arise de novo. As I will argue, they develop out of already existing ideologies. Consequently, I frame each narrative in relation to the ideology espoused by the projectivity of its author; namely, the capitalism of Adam Smith, the idealism of Immanuel Kant, and the pragmatism of John Dewey, respectively. In light of the different trajectories constructed through the projectivity of Gates, Stallman, and Torvalds, I refer to the corresponding narrative of each of these spokespersons as the entrepreneurial narrative, the idealistic narrative, and the pragmatic narrative.
Figure 4 (see next page) illustrates the competing narratives constructed by these spokespersons in the process of not only imagining different ideologically-encoded trajectories of action for the software programming culture but also reaffirming the shared practices of the culture (e.g., none of these narratives constructs a future that advocates doing away with any of these iterative practices and, as a result, reifies those very practices and one of the processes by which these practices is reactivated). In my discussion, I emphasize the primary, though not necessarily singular, trajectories for achieving each author’s construction of democratic freedom, and I analyze the ways in which the ideologies embedded in software and its related practices create always competing, though not always antithetical, sets of relations in the software programming culture. And finally, by examining the ways in which the discourse of these spokespersons has remained consistent over time, I illustrate how the construction of these narratives have evolved. In doing so, I illustrate for lay users of software the ways in which the discourse of software is encoded through “the intervention of ideology into language” (Grossberg, 1996a, p. 137).

The Entrepreneurial Narrative of Gates and the Professionalization of PC Hobbyists

Modern conceptions of entrepreneurialism are rooted in the free market, which was first conceptualized in 1776 by Adam Smith, the founder of modern laissez-faire economics, in his seminal work *The Wealth of Nations*. Writing at a time when a number of nations were evolving from an agrarian-based economic system to a manufacturing-based economic system, Smith posits that wealth consists not singularly in the riches of landowners but “in the consumable goods annually reproduced by the labour of the society” (p. 862). These goods function as commodities, the value of which “to the person who possesses it, and who means not to use or consume it himself, but to exchange it for other commodities, is equal to the quantity of labour which it enables him to purchase or command” (p. 44). With the emphasis on materialism through commodity production and consumption, the market takes precedence in Smith’s system: the cost of bringing a commodity to market coupled with the
Figure 4: The Ideological Encoding of Software and Its Practices through Narratives
market’s demand for that commodity generates wealth, which in a market economy is money, “the common instrument of commerce” (p. 45).

Integral to the success of the market are those individuals who are willing to invest their own capital in the identification and development of commodities, a process that Smith likens to an “adventure” (p. 68). But because this adventure involves risk, Smith contends that those who undertake such risk—persons whom we now typically refer to as entrepreneurs—do so for their own economic interests: “But it is only for the sake of profit that any man employs a capital in the support of industry; and he will always, therefore, endeavor to employ it in support of that industry of which the produce is likely to be of the greatest value, or to exchange for the greatest quantity either of money or of other goods” (p. 572). Thus, the aim of the entrepreneur is to maximize her investment, which is measured by the accumulation of “capital” accrued through the unfettered relationship between producer and consumer in that market.³

However, entrepreneurial “self-interest,” also referred to as “self-love” (p. 24), is not without benefit to others, according to Smith. With the accumulation of capital, the entrepreneur is able to hire other “industrious people” for their labor, a process that serves to increase the speed at which a given commodity is produced, thereby increasing the speed at which said commodity can be consumed (p. 68).⁴ This “division of labour” advances not only the entrepreneur but also the worker and the poor. The worker, or laborer, is able to gain employment by which to earn a wage that, in turn, allows her to pay rent, buy food, and purchase products. The poor to whom “the rich is made to contribute…relief” (Smith p. 68) are more likely to be supported because of the increased opportunities for new wealth generated through entrepreneurialism. Thus, the market, with its corresponding benefits for all individuals, whether the entrepreneur, the laborer, or the poor, is guided by what Smith describes as an “invisible hand,” an often, though not unproblematically, invoked metaphor.

³ My use of “unfettered” here stems from Smith’s assertion that government interference in the free flow of commodities disrupts the economic system itself by inhibiting competition (pp. 572-576). However, for Smith the only thing worse than government disruption is the monopoly of a specific market by a particular company (pp. 86-87, 622-624, 771-783).

⁴ Smith uses the manufacturing of pins to illustrate the effectiveness of the “division of labour” (pp. 10-11). One worker with little knowledge of the modern means of pin-making “could scarce, perhaps, with his utmost industry, making one pin in a day, and certainly could not make twenty.” However, because of the division of labor in the pin-making industry in which “one man draws out the wire, another straightens it, a third cuts it, a fourth points it,” as few as ten men can make “upwards of forty-eight thousand pins in day.”
used in modern economics to describe the beneficial connection between economic self-interest and the public good.

Because the pursuit of one’s “own interest…frequently promotes that of the society more effectually than when he really intends to promote it,” (p. 572) Smith contends society as a whole benefits from “every individual…continually exerting himself to find out the most advantageous employment for whatever capital he can command.” Thus, economic enterprise stoked by a free market as well as the production and consumption that occurs through it generates revenue through all those who labor; wealth then is generated by the production and consumption of every single individual and distributed among all rather than a few. This move away from the view of wealth in the hands of a few to the labor of all is the birth of democracy through economics. Consequently, in a “free-market” democracy, freedom arises from the ability to produce, consume, and thus generate wealth through the market that constitutes for Smith “a system of perfect liberty” in which producer and consumer are “left perfectly free to pursue [their] own way” (p. 572).

Some of the most successful entrepreneurs in the modern history of the United States include George Eastman, of Eastman-Kodak; Thomas Edison, of Commonwealth Edison; Henry Ford, of the Ford Motor Company; and J.P. Morgan, of Morgan-Stanley Banking. Joining the ranks of these premier entrepreneurs is Bill Gates, co-founder and, since 2003, Chairman and Chief Software Architect of Microsoft, the world’s most successful software company. Like the entrepreneurial pioneers who came before him, Gates helped to pioneer

---

5 Smith’s use of the “invisible hand” in _The Wealth of Nations_ reads as follows:

[Every individual] generally, indeed, neither intends to promote the public interest, nor knows how much he is promoting it. By preferring the support of domestic to that of foreign industry, he intends only his own security; and by directing that industry in such a manner as its produce may be of the greatest value, he intends only his own gain, and he is in this, as in many other cases, led by an invisible hand to promote an end which was no part of his intention. Nor is it always the worse for the society that was no part of it. By pursuing his own interests he frequently promotes that of the society more effectually than when he really intends to promote it. I have never known much good done by those who affected to trade for the public good. (p. 572)

Readings, or rather, misreadings of Smith’s use of “invisible hand” in modern economics to support the idea that individual economic interests serve the public as a whole have recently been challenged on a number of fronts with most challenges stemming from the decontextualization of the metaphor from the texts and time in which Smith was writing in applications of contemporary discussion of market economics (Grampp, 2000; Persky, 1989; Rothschild, 1994, 2001). For example, Rothschild (1994) suggests that Smith’s use of the “invisible hand” was “ironical” and 20th-century usage of the simile to connect individual interests with public good through the market “is the sort of idea [Smith] would not have like (p. 319). And Persky insists that Smith’s anti-foreign investments stance makes the connection between personal interest and social welfare in the context of international trade more complicated than commonly noted in discussions of the “invisible hand.” However, my use of the “invisible hand” in my discussion of the entrepreneurial narrative reflects the commonly held economic view that self-interest serves the public good via the “invisible hand” because it is a commonly held view in the exaltation of market economics.
an industry—the proprietary software industry. As with most pioneering enterprises, the beginnings were humble and somewhat tenuous as his entrepreneurial narrative of software reveals.

In February 1976, Gates, as a general partner in the newly-formed proprietary software company “Micro-Soft” which in 1975 had revenue in the amount of $160,000, wrote “An Open Letter to Hobbyists,” which was first distributed throughout the country via the newsletter “Computer Notes,” published by the Homebrew Computer Club, a computer hobbyist organization. In this narrative, Gates constructs the events of how he and his partner, Paul Allen, came to develop a version of the programming language BASIC for the Altair—what he and Allen called Altair BASIC—only to have the software, according to his version of this story, stolen by computer hobbyists. Gates’ projectivity is manifest through this narrative that seeks to direct software development away from the hobbyist practice of sharing software and towards the construction of software as intellectual property with economic value that must be guarded. Through his creative response to an emergent situation in the nascent PC software programming culture, Gates imagines a future rooted in non-redistributable black-boxed software developed by independent PC software vendors owned by entrepreneurs, who, in order to increase their capital, would charge a fee for the restricted use of their commodity.7

Because narrative is intricately tied to the process of identity-building, we can understand the story of Altair BASIC as socially positioning software entrepreneurs and

---

6 BASIC, or Beginner’s All-Purpose Symbolic Instruction Code, is a programming language originally developed for mainframe computers and later modified for mini and microcomputers, as was the case with Altair BASIC. In order to create software for computers, programmers need tools such as programming languages to create operating systems, applications, as well as other tools.

7 Most software before the PC era was developed gratis for business and industry either in-house or by hardware companies such as IBM, whose revenue came primarily from computer hardware sales and leasing. In 1969, when mainframes still reigned, IBM decided to “unbundle” its software from the hardware it manufactured, meaning that software had to be purchased separate from hardware, which at the time was purchased or, more likely, leased by institutions and organizations. Because of this decision, which was motivated at least in part by the antitrust lawsuit that had been filed against IBM by the U.S. government due to IBM’s dominance in the market of mainframe computers, software came to be seen as an economic product in its own right rather than a free package that came with the hardware (Ceruzzi, Campbell-Kelly). IBM, whose revenue continued to derive primarily from hardware sales, particularly mainframes and minicomputers, became the largest and most profitable supplier of software products (*Datamation*). However, IBM was by no means the only supplier. Software was and continues to be developed by hardware companies such as IBM, Unisys, Siemens, HP; by in-house programmers who create software applications for specific organizations such as banks and accounting firms; by academic and government institutions; and by independent software vendors such as Microsoft, Oracle, and Lotus (Campbell-Kelly, pp 124-131). By 1998 Microsoft had succeeded IBM as the number one supplier of software in the world. With the development of the PC, which made individual ownership of computers possible, hobbyists, who are typically referred to as hackers nowadays, stood poised to become another software supplier.
those involved in the capitalistic enterprise of software development against those who share software freely. Not surprisingly, the social positioning constructed through Gates’ narrative involves issues of power constituted through rhetorical configurations of good and evil as they function specifically in the software programming culture. Entrepreneurs are constructed as both pioneers and benefactors who in their exercise of freedom generate new economic wealth by producing much needed commodities for those who wish to exercise their freedom to consume—the PC software market, in general, and hobbyists, more specifically. Hobbyists who share software, on the other hand, are constructed as thieves who deny entrepreneurs the freedom to accumulate capital and to employ laborers for the development of software, thereby denying programmers an opportunity to earn an income. In addition, according to Gates’ narrative, hobbyists also threaten the viability of the market itself, which, in turn, threatens the freedom of those who have a desire to consume software legally. In short, hobbyists threaten the democratic freedoms that stem from the free market, unlike entrepreneurs who encourage such freedoms through the commodification of software, as the following analysis illustrates.

**The Narrator as Entrepreneur, Benefactor, and Victim**

In “An Open Letter to Hobbyists,” Gates recounts how he and Allen recognized a new opportunity in software with the birth of the PC; he writes, “Almost a year ago, Paul Allen and myself, expecting the hobby market to expand [with the development of the Altair 8800], hired Monte Davidoff and developed Altair BASIC.” Like many stories of American entrepreneurialism, Gates’ narrative does not begin with high-powered venture capitalists, boardrooms, and CEOs but with two struggling entrepreneurs, their one employee, and their kairotic recognition of an untapped market opportunity that with their investment of time and money would meet the needs of consumers. Meeting these needs with a proprietary commodity—Altair BASIC—would, they bet, generate a profit that would allow them as entrepreneurs to develop more and better products and to employee more programmers. Explaining the process of development to his readership, Gates writes, “Though the initial work took only two months, the three of us have spent most of the last year documenting, improving and adding features to BASIC. Now we have 4K, 8K, EXTENDED, ROM and
DISK BASIC. The value of the computer time we have used exceeds $40,000.” Gates recounts the development, documentation, and improvements to Altair BASIC in economic terms. The apparent economic investment made by Gates and Allen explains how the documentation, improvements, additional features, and versions of Altair BASIC came about. It is a return on this investment through monetary payment—the “self-interest” of the entrepreneur to use Smith’s characterization—that makes future software development and improvements possible, according to Gates’ narrative.

The price of hardware and the software that ran on it had been unaffordable to individual hobbyist users until the PC.⁸ Because the Altair 8800 with its Intel 8080 microprocessor was marketed for affordability⁹—only $400 for the first kits—the computer market was opened up to a greater number of possible users than ever before, most notably hobbyists. Gates and Allen priced the use of a proprietarily-owned, non-redistributable copy of Altair BASIC without the source code between $60 and $500, depending on the version.¹⁰ By pricing their software, albeit reasonably and “at special low prices for Altair customers,” as advertised in a MITS’ newsletter article “Altair BASIC—Up and Running,” Gates and Allen constructed their product as a proprietary commodity with its most valuable asset, the source code, black-boxed, meaning that the source code was not made available to users. In short, source code had value that had to be protected. This construction was particularly significant because hobbyists in particular were accustomed to sharing products, especially software, the price of which had made the practice of sharing a necessity. Before the PC,

---

⁸ Altair BASIC was not developed on an Altair but on a PDP-10 mainframe computer at the Harvard University Computing Center. Because Gates and Davidoff were Harvard students at the time, the PDP-10 was available to them and they did not have to pay for time on the computer, which was a common practice in the age of mainframe computers. Consequently, the $40,000 of computer time that Gates references in his narrative is not a monetary investment that Gates and Allen had actually made but an assessment of the value of the computer time should they have had to pay. Interestingly, Harvard had a policy that the PDP-10 was not to be used by students for commercial purposes (Ceruzzi, p. 235).

⁹ For example, IBM’s first software product after its decision to unbundled its software was the Customer Information Control System, which IBM licensed for $600 per month (Mounce, 2003).

¹⁰ MITS, the manufacturer of the Altair 8800, even created a monthly-payment plan option that allowed individual users, as the advertisement stated, to purchase the computer “without taxing your pocketbook” (MITS, Kit-a-Month). With each payment, users were sent monthly a piece of the kit until they acquired all the pieces necessary to put the Altair together. An option such as this helped MITS to achieve their goal of “making computer power available at a price most everyone can afford.”

¹¹ Different versions of Altair BASIC were written so as to meet different memory capabilities of the hardware. The more memory available the more feature that could be included. The 4K version of BASIC was priced at $60, the 8K at $75, the extended version at $150. Users who wanted BASIC adapted to Intel 8080 microprocessor using-hardware other than the Altair were charged $500 (MIT, “Altair BASIC—Up and Running”).
software had either been included in the price of the hardware or been priced for purchase by
government organizations, academia, and corporations who could afford both the software
and the hardware on which it ran. Consequently, hobbyists had developed out of necessity
the habit of sharing copies of software as well as the software’s source code if it was
available. Gates’ narrative challenges this practice in the software programming culture by
asking the rhetorical question, “Hardware must be paid for, but software is something to
share?” As with most electronic innovations, those most enthused by the prospect of owning
and using the extremely limited-capability Altair 8800, which was programmed by flipping
switches that then made the lights on the hardware blink in a particular order, were hobbyists,
the audience for Gates’ narrative. By locating the value of Altair BASIC in economic terms
rather than in hobbyist interest and enthusiasm, Gates clearly designates software for the
developing PC software programming culture as a non-redistributable proprietary product,
the use of which demanded monetary compensation; source code, which is never mentioned
in Gates’ narrative because Micro-Soft did not release the source code for Altair BASIC, is
thus constructed as a resource that must be protected from users by independent software
vendors through the practice of black-boxing.

As such, the development of Altair BASIC by Gates, Allen, and Davidoff is
constructed as “professional work” in this story, not the hobbyist tinkering of amateur
programmers. Gates asks, “What hobbyists can put 3-man years into programming, finding
all bugs, documenting his product and distribute for free?” The benefit of identifying
software-related activities as a professional enterprise is supported, according to Gates, by
the quality of Altair BASIC, as evidenced by the feedback received from those using this
programming language; Gates writes, “The feedback we have gotten from the hundreds of
people who say they are using BASIC has all been positive.” According to this narrative, to
accomplish the tasks demanded in the creation of quality software and its accoutrement,

---

12 For example, the Mark IV, a file-system management program for the popular mainframe IBM System/360 sold for
$30,000 in 1969 (Campbell-Kelly, pp. 106-107). Individuals obviously could not afford the software let alone the hardware
it ran on. But individuals who had access to computer through work or school and wanted software for their own use had
learned to share programs. Sharing as a viable practice had begun as early as 1955, when customers of the IBM 701
mainframe computer were facing an upgrade to the 704. Instead of creating software independently of one another, users
decided to band together in a user group entitled SHARE, thereby defraying as much as possible the very expensive cost of
developing software by sharing information, software tools, and applications with one another (Campbell-Kelly, Ceruzzi).
After the first year, sharing through SHARE, which included organizations from the U.S. and Europe, saved members
approximately $1.5 million (Campbell-Kelly, p. 33). Development groups such as SHARE became relatively commonplace
after this.
software development must be approached as a professional enterprise and all that is entailed in such an enterprise: an entrepreneurial investment of time and money, the employment of programmers, the proprietary ownership of software, and the black-boxing of source code. The amount of time and money that hobbyists put into developing software tools and applications can never produce the quality products that professionals can. Consequently, hobbyists’ work on or, more accurately, play with software development and the production of quality software are constructed as mutually exclusive. On the other hand, professional work on for-profit software by professionals and the production of quality software are mutually inclusive in this narrative. After all, the lack of professional work on PC software had led to the situation that the entrepreneurs are responding to and trying to remedy, as Gates points out—“To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software and an owner who understands programming, a hobby computer is wasted.”

But the entrepreneurial approach to software development constructed by Gates is hardly one motivated only by economic self-interest, according to this story’s narrator. Gates and Allen are not looking to get rich from their product: “the royalty paid to us, the manual, the tape and the overhead make it a breakeven operation.” Instead, Gates constructs these entrepreneurs as benefactors to a market in much need of quality products. According to this narrative, what motivates these capitalist-driven entrepreneurs is satisfying the need for quality software in a market by giving hobbyists the freedom to consume it, not profiting from the product’s sales. Without entrepreneurs such as Gates and Allen responding to this gap and possible crisis, the hobbyists’ freedom to find the products they need in the market, as well as the very existence of hobbyists in the culture of software programming, is threatened.

However, the entrepreneurial desire to do good is threatened, according to this narrative, because Gates and Allen have not received a return on their investment. Gates writes, “The amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC worth less than $2 an hour.” Because quality software demands professionals dedicated to the full-time tasks of development, users must give the entrepreneurs who create the commercial-quality products they use their due by paying for
use of the software; to not to do is to deny entrepreneurs their democratic freedom to generate capital from their work. To fight this denial of their freedom, programming professionals will be forced, according to this narrative, to deny consumers their freedom by withholding the very thing that the market needs most—quality software. As Gates suggests in a not-so veiled threat, “there is very little incentive to make software available to hobbyists,” if the relationship between producer and consumer, a relationship that lies at the core of entrepreneurialism, is not respected. Gates and Allen, through the production of Altair BASIC, are trying to fulfill the needs of a market and exert their freedom as producers, but the should-be consumers—hobbyists—are making this capitalistic enterprise nearly impossible because of their failure to recognize software products for the PC as a commodity with monetary value, according to this narrative. Consequently, Gates argues that hobbyists who do not pay for the use of software such as Altair BASIC are subverting the freedoms made possible to both producers and consumers by the market.

**The Hobbyist Audience as Thieves or Police**

In Gates’ “An Open Letter to Hobbyists,” the audience of hobbyists is constructed as amateur programmers and as such are juxtaposed to professional programmers. However, not only are hobbyists relegated to amateur status, many are also constructed as thieves, from whom software must be protected. Hobbyists who do not pay for their software are doing more than failing to uphold their role in the producer/consumer relationship; these hobbyists threaten the continued existence of the hobbyist market by denying entrepreneurs the freedom to generate a profit from their commodities. Consequently, hobbyists are constructed as criminals. As Gates succinctly states, “Most directly, the thing you do is theft.” The sharing, or redistribution, of software among users, a relatively common pre-PC practice, is suddenly criminalized through the rhetorical construction of the “An Open Letter to Hobbyists.” Because software is a commodity resulting from the investments of entrepreneurs and the work of professional programmers, software, without any dependence on the hardware on which it runs, is constructed as the source of economic value in this story. To take a product with such value without paying for it is stealing. Thus, hobbyists who share are constructed in Gates’ narrative as thieves. And, those who are stolen from—Gates and Allen as well as Micro-Soft—are consequently victims of a crime. Because of the way in
which Gates constructs this story, the crime is made particularly abhorrent because the victims were trying to help those they were victimized by—hobbyists.

The victimization of entrepreneurs such as Gates and Allen through the stealing of proprietary software, what was later constructed as “piracy” because of the theft of intellectual property, is constructed as commonplace in this narrative. Gates writes, “As the majority of hobbyists must be aware, most of you steal your software.” And even those who do not “steal” condone this thievery by turning a blind eye to the theft or by choosing not to recognize the criminality inherent in so-called sharing. To counter the growing criminal state of the hobby market, Gates argues that it is up to hobbyists to police one another to prevent stealing. Though the idea of hobbyists policing hobbyists might have stunned the audience as such a practice had never occurred before, Gates claims that policing is in fact already taking place without any encouragement from the entrepreneurs of Altair BASIC. Others who have no investment in this programming language have found this practice to be criminal and have, according to Gates, “reported [guilty hobbyists] to us…” To punish software thieves and ensure the freedom of hobbyists and entrepreneurs alike, Gates constructs what he considers to be an appropriate punishment but one that can only succeed by hobbyists policing hobbyists—kicking thieves “out of any club meeting they show up at.” If criminal hobbyists are going to deny freedom to others, then the only thing to do is to deny them the freedom to participate in social activities relating to their hobby. As Gates reminds his readers, hobbyists who steal are not only committing a crime against software entrepreneurs and producers but also “giv[ing] hobbyists a bad name.” Through his narrative construction, Gates makes hobbyists who are not stealing software invested in the outcome of this situation by suggesting that not only is the freedom to consume threatened but also the good name and reputation of those hobbyists who uphold the democratic freedoms of production and consumption by not sharing software.

A Story of Good vs. Evil

The story of the development, distribution, and theft of Altair BASIC, as constructed by Gates, is not just the story of a software program or even of entrepreneurialism. The story of Altair BASIC is, in fact, a story of good vs. evil. On the side of good are Gates and Allen, entrepreneurial pioneers, who invest their time and money in an effort to meet the unmet
needs of a market suffering from the lack of good software, who create employment opportunities for professional software developers, and who want to continue to create good software by reinvesting into research and development (R&D) what they make from the sale of the use of their commodity, all of which are freedoms available to them through the free market. According to Gates’ narrative, without entrepreneurs who are willing to do these things, the hobbyist market would continue to suffer from the inability of hobbyists to exert their freedom to consume quality products.

On the side of evil in the “An Open Letter to Hobbyists” are those hobbyists who steal their software by sharing or condone thievery by not doing anything to prevent or punish those who steal. Both groups of hobbyists make victims out of the very entrepreneurs and professional programmers who want to supply the hobby market with what it needs most—quality software—by denying entrepreneurs the freedom to garner a profit from their products and programmers the opportunity to earn a wage through their labor. These behaviors, according to Gates’ narrative, give a bad name to all hobbyists because hobbyist consumers as a whole are constructed either as ungrateful thieves, who steal that which they should pay for and do not recognize through payment for use a commodity the risks that entrepreneurs make in an effort to bring good software and its accoutrement to the hobbyist market, or as accomplices to this crime because of their turning-a-blind-eye to the illegal activities of their associates.

Consequently, the theme of good vs. evil, in Gates’ narrative, as a rhetorical construction of his projectivity, sets for the PC software programming culture a trajectory for future action that reifies the primacy of the economic market (i.e., capitalism) and the freedom of individuals to produce and consume commodities in the democratic generation of wealth, in a process similar to that described by Smith. Although software development certainly occurred before the birth of the PC software programming culture, software had not typically been recognized by many academics, researchers, and hobbyists as something they had to pay for either because the software had little if any economic value in its own right as it was considered a component of hardware, which is where the value really laid, or the price of the software was so high as to be out of reach of the individual user. Gates, through the agency manifest through his projectivity, destabilizes this view and envisions for the
software culture a future that is firmly rooted in non-redistributable, black-boxed proprietary software produced by professionals for consumers.

To be democratic, to be lawful, to be good is to pay for the use of commodities needed by a market and developed by entrepreneurs who undertake in their exercise of freedom substantial risk by investing their time and money in the belief that they can meet the needs of a market, thus making the market itself more economically productive and generating wealth for the health of the nation. Recognizing the value of the labor invested in a product is accomplished in the market through payment of monies equal to the economic value of the product. Not to give recognition through payment is not only unlawful but threatens the democratic underpinnings of society, which in this narrative is deeply embedded in notions of entrepreneurialism, the market, and the primacy of the producer/consumer relationship. If such illegal and undemocratic practices continue in the burgeoning PC-software market, according to Gates’ narrative construction, the market itself could collapse, thus depriving hobbyists of a much needed product to consume, professional programmers of employment opportunities, entrepreneurs of a market through which to accumulate capital from their products, the nation of wealth generated through the labor of all its citizens, and finally, every individual their inalienable right of democratic participation in and through the market.

Within three years of Gates’ narrative, the “gold rush” for PC software development had begun. Gates success in the technological and economic fields is well documented. However, in the field of discourse, his success lay in re-imagining what was then nothing more than a hobbyist interest and what this re-imagining might achieve. Rhetorically, he succeeded by introducing a narrative of entrepreneurial capitalism into what was then only a hobbyist practice of PC software programming, meant to indulge their nerd/geek technophilia. Through the narrative construction of “An Open Letter to Hobbyists,” Gates professionalized the PC software programming culture at its earliest beginnings, and this discursive success underpins the technological and economic success of what we know now as established software technology.
Another conceptualization of democratic freedom—one that runs counter in many ways to the democratic freedom constructed through entrepreneurialism—is idealism. By idealism, I mean a philosophy that asserts all action should be guided first and foremost by a moral purpose. Kant’s “categorical imperative,” or universal law of morality, is one articulation of this idealism. Generally referred to as transcendental idealism, Kant believes that all individuals should “act that you use humanity, whether in your own person or in the person of any other, always at the same time as an end, never merely as a means” (1998, 4:429). If given the freedom to act according to their reason, Kant argues, individuals ought to act morally and ethically so that the interest to act is motivated by the following maxim: “I ought to do so and so, even though I should not wish for anything else” (1988, 5:31). Thus, acting according to one’s will, which is determined by one’s reasoning according to the categorical imperative, is independent of immediate material or practical effect and instead is the result of rationality, which, in the case of Kant, dictates that one act morally so that “the highest good [is] a good common to all” (1988, 6:97).

A necessary precondition to exerting one’s will is democratic freedom, according to Kant, because it is only through truly democratic means that the freedom to act morally exists. As such, democratic freedom for Kant is the ability to do one’s “duty” by acting honestly, beneficently, morally, and with dignity according to one’s will, a process which in turn “forms a kingdom of ends” in which humanity is in harmony in its cooperative quest for the transcendental ideal. To hinder or prevent another’s ability to act according to her reason and will is consequently an “attack on freedom.” As Kant states, “For then it is clear that he who transgresses the rights of men intends to use the person of others merely as a means, without considering that as rational beings they ought always to be esteemed also as ends, that is, as beings who must be capable of containing in themselves the end of the very same

---

13 My use of “truly democratic means” here is a nod to the difference between Kant’s use of “democracy” and the modern usage of the term. In *Perpetual Peace*, Kant actually takes issue with democracy because of the tendency of those in power to assert their on will at the expense of the will of others: “Of the three forms of the state, that of democracy is, properly speaking, necessarily a despotism, because it establishes an executive power in which "all" decide for or even against one who does not agree; that is, "all," who are not quite all, decide, and this is a contradiction of the general will with itself and with freedom” (p. 34). To combat this form of democracy, Kant argues for republicanism. However, Kant’s description of a republic mirrors the modern conception of democracy. As John Bacher notes, “When he refers to ‘republicanism’ he means what we now call liberal democratic systems of government that ensure majority rule through elections, while at the same time respecting the basic human rights of minorities…division of powers and a constitution” (p. 20).
action” (*Fundamental*, p. 3:27). Though Kant himself was dismissive of the maxim, “Do unto others as you would have done to you,” a number of scholars have subsequently concluded that a morally-driven reading of this maxim embodies Kant’s categorical imperative and is, in fact, a popular representation of the idealism found in Kantian discussions of morality and democratic freedom.\(^{14}\) It is, I argue, a similar representation of idealism functioning as a guide for the future actions of programmers that Stallman imagines in his narrative in which he calls for programmers to act morally in the development and use of software by following the golden rule. Through his projectivity, Stallman’s narrative is intricately tied to both the PC era and the period of the “hacker ethic” that predates the PC.

In September of 1983, a mere seven years after Gates’ “An Open Letter to Hobbyists,” Stallman, of the MIT Artificial Intelligence (AI) Lab\(^{15}\), posted “new UNIX implementation” to a UNIX newsgroup, net.unix-wizards newsgroup, via USENET, a distributed bulletin board system that predates the Internet. In this post, Stallman announces his “GNU Project,” constructs the story of why he decided to create “Free Unix”\(^{16}\) and solicits help in making this project happen. “GNU,” a recursive acronym\(^{17}\) that means GNU’s Not UNIX, was intended to offer a free alternative to UNIX. At the time, UNIX, a commercial proprietary OS developed at and inexpensively licensed by Bell Laboratories of AT&T\(^{18}\), was the most widely used OS in business, industry, and academia. UNIX was unique for two reasons: first, UNIX was written in the programming language C which made

\(^{14}\) In a footnote in his *Groundwork of the Metaphysics of Morals*, Kant, referring to a negative variation of the golden rule—Do not do to others what you do not want done to you—considered this maxim “trite” and without the necessary underpinning of moral duty (4:430). Consequently, a murderer, according to Kant could reason that because he would not mind being murder that he could murder. However, scholarship on the Golden Rule points to the need to understand this maxim as a morally grounded (see Thomas, 1970; Weiss, 1941) rather than relativist. In fact, Weiss, writing in the *Journal of Philosophy*, contends that Kant’s imperative “is a good translation of the Golden Rule” (p. 430).

\(^{15}\) The Artificial Intelligence Lab at MIT was the heart of the “Tech Square Monastery” at the MIT campus (Levy, p. 415). Stallman joined the lab in 1971. Until the 1990s when the Silicon Valley, along with Stanford University and University of California at Berkeley, became the hub of software development, Boston, in large part because of MIT and Harvard University, had been the software mecca (Ceruzzi, p. 140).

\(^{16}\) Although Stallman, as well as others, uses ‘Unix,’ the actual trademark is ‘UNIX.’ Consequently, I use ‘UNIX’ except when quoting from a text in which the author uses ‘Unix.’

\(^{17}\) Recursive acronyms such as “GNU’s not Unix” are a common language hack among technophiles in the software programming culture. And *The New Hacker’s Dictionary* makes the connection between the language play of recursive acronyms and technophilia in the following comment: “A hackish (and especially MIT) tradition is to choose acronyms/abbreviations that refer humorously to themselves or to other acronyms/abbreviations (Raymond, 1996, p. 382).

\(^{18}\) AT&T, which was the initial developer and owner of UNIX, had at the time a regulated monopoly of telephone services. To keep this monopoly, AT&T “had agreed not to engage in commercial computing activities,” (Ceruzzi, p. 282). Consequently, AT&T licensed UNIX for a “nominal” fee in accordance with its agreement with the U.S. government.
it the first portable OS, meaning that it was not specific to a particular type of hardware and was consequently cross-hardware compatible to any computer that had a C compiler; and second, UNIX’s source code was made transparent—a rarity with proprietary software— by its owner, AT&T, rather than black-boxed, a fact that allowed programmers and other IT professionals to study the code, to modify it to their systems and applications, and to redistribute both the software and any modifications. However, AT&T’s decision to release UNIX’s source code did not negate the fact that UNIX was proprietary and could be black-boxed or, even more likely, assigned a hefty use-fee at any time. Consequently, in order to make what Stallman considered to be a truly free UNIX-compatible OS that could never be black-boxed and that also shifted monetary value away from the code itself, he began the GNU Project.

Unlike UNIX, Stallman’s GNU Project would be “free,” a fact that contrasted sharply with the dominant entrepreneurially-driven software development that Gates and others like him had made dominant by the time Stallman wrote his post in 1983. Consequently, Stallman’s e-mail is more than a descriptive story of his decision to create the GNU Project; this narrative attempts to disrupt the entrenchment of the software programming culture in the capital motive of the market. He attempts to destabilize the ideological trajectory of Gates’ narrative that constructs code as an entrepreneurial commodity by which to accumulate an economic profit through the sale of non-redistributable, black-boxed software.

---

19 The release of UNIX’s source code was unique. As Ceruzzi points out, “most computer vendors guarded source code as their family jewels, seldom gave it out, and did all they could to lock a customer into their products” (pp. 284-285). The fact that UNIX’s source code was made transparent has led at least one historian of software to call UNIX “a non-proprietary system” (Campbell-Kelly, p. 143) when, in fact, it is proprietary. (The transparency of UNIX’s source code led to a phenomena known as Though the significance of its proprietary status may have seemed inconsequential given that UNIX had always been transparent, the lawsuit against IBM for alleged copyright infringement of UNIX’s “protected code” in 2003 by UNIX’s current owner—the SCO Group, which acquired UNIX in 1995 from Novell—illustrates the potential danger of confusing transparent code with free or open software.

20 One example of the adaptation made possible through the transparent release of UNIX’s source code is Berkeley UNIX, also known as BSD. BSD, adapted and modified from UNIX’s source code, was made proprietary by SUN Microsystems. Unlike AT&T, which never generated a profit from UNIX even after its divestiture and in spite of the fact that UNIX was by 1991 the most widely used OS in the world, SUN reaped hefty profits from BSD (Ceruzzi, p. 285).

21 The Hacker’s Dictionary includes an entry “UNIX conspiracy,” which details an alleged plot by AT&T’s Bell Laboratories to make potential competitors dependent upon AT&T. “This would be accomplished by disseminating an operating system that is apparently inexpensive and easily portable [because UNIX was used the C programming language], but also relatively unreliable and insecure (so as to require continuing upgrades from AT&T)” (p. 461). Of course, the fact that AT&T later sold UNIX to Novell in the early 1990s dispelled this conspiracy, as the Dictionary acknowledges. However, I contend that an unforeseen consequence of the release of a proprietary OS’s source code, as in the case of UNIX, was the increased possibility of its code being used in other OSs. It is this consequence that lies at the heart of SCO’s lawsuit against IBM.
Consequently, Stallman’s story “new Unix implementation” reads as a narrative of resistance constructed by an idealist who, through his projectivity, seeks to direct the future of the software programming culture away from the singular pursuit of economic capital through proprietary software and towards the idealism of the “golden rule.” In doing so, Stallman positions himself and others like him in opposition to entrepreneurs such as Gates. Because this social positioning is an inversion of the sets of relations created by the entrepreneurial narrative, Stallman’s idealistic narrative also invokes the theme of good vs. evil; however, those on the side of evil and those on the side of good are inverted, as the following analysis illustrates.

**The Narrator as Idealistic Hacker, Revolutionary, and Victim**

The beginning of Stallman’s story lies in events not developed in detail in the narrative itself, most likely because his audience of UNIX programmers, as was Stallman himself, would have been familiar with the current state of the culture. At the time that Stallman constructed his narrative—September 1983—the PC software programming culture had been fully enveloped in an entrepreneurial approach to software development as a result of what one computing historian has identified as the proprietary software “gold rush,” which took place from 1979 to 1983 (Campbell-Kelly, 2003, p. 203). The rush to create and release non-redistributable, black-boxed software had created a boon not only for software entrepreneurs but also for the national economy, which at the time was just starting to eek its way out of the major recessions of the early 1980s.²² Because of the commodification of software, which necessitated non-redistribution and black-boxing, the PC software industry stood poised to generate revenues of almost $1 billion (Campbell-Kelly, 2003, p. 96) in the year in which Stallman wrote his post “new UNIX implementation.”

For Stallman, the monetary rewards born of proprietary, non-redistributable, black-boxed software, by then an established technology, were not an unqualified good, as his narrative explains. Stallman believed, in fact, that charging for the use of non-redistributable

---

²² The National Bureau of Economic Research, the U.S.’s leading nonprofit economic research organization, defines a recession as “a significant decline in economic activity spread across the economy, lasting more than a few months, normally visible in real GDP, real income, employment, industrial production, and wholesale-retail sales” (www.nber.org/cycles/recessions.html). The two recessions of the early 1980s were marked by a significant decline in the GDP, soaring inflation, and high unemployment. Rare growth industries were found in the service sector of the economy, and out of these industries computer and data-processing was ranked the highest (Plunkert, p. 10).
software “guarded” by black-boxing was morally wrong. Consequently, he imagined through his “new UNIX implementation” an alternative to the reigning entrepreneurial paradigm that closed-up source code and made software non-redistributable so that monetary profit could grow from the code itself, a process that had led to the creation of vast economic capital for both software entrepreneurs and the U.S. economy. Stallman is thus constructed in this narrative as a revolutionary who desires to overturn the stranglehold of the profit-motive through the object of software in the software programming culture and seeks to resurrect a dying, if not dead, approach to code development through the sharing of redistributable and transparent source code that allowed for modification and redistribution. The practices of sharing software, demonized by Gates’ narrative, is rooted for Stallman in the morality of the golden rule and in the hacker ethic—the moral code of programming that he as well as many others had lived by in the pre-PC world.

As Levy (1994) explains in his seminal work *Hackers: Heroes of the Computer Revolution*, which is about programming from the 1950s to the early 1980s, followers of the hacker ethic believe, among other things, that

- access to computers—and anything which might teach you something about the way the world works—should be unlimited and total.
- all information should be free.
- computers can change your life for the better.23

In accordance with these beliefs, Stallman sought through the projectivity manifest in his narrative to resurrect the practice of sharing software and source code, a practice that before the birth of the PC had been relatively commonplace, especially among research and academic hackers but that had since then been aggressively attacked through discourse that equated sharing with theft. With his announcement “Free Unix!,” Stallman constructed a trajectory for future action in the software programming culture that moved away from the production and consumption of commodified software that necessitated the protection of that software from users and towards democratic freedom, not the kind of freedom constructed

---

23 Other elements of the hacker ethic include the following: “Mistrust Authority—Promote Decentralization…, hackers should be judged by their hacking, not bogus criteria such as degrees, age, race, or position…, you can create art and beauty on a computer. (pp. 41-46). I chose to emphasize issues of access, freedom of information, and quality of life in the discussion above because these issues are what most clearly fuel this particular narrative by Stallman. However, subsequent narratives by Stallman reinforce these other elements of the hacker ethic.
through entrepreneurialism, but the kind of freedom embodied in the idealism of the hacker ethic as well as the golden rule.

Because the black-boxing of source code was a technique of the entrepreneurial commodification of established software technology that necessitated releasing source code only in machine-readable language (i.e., binary code made up of ones and zeros series of 1s and 0s), a process that prevented the modification of the code, Stallman’s narrative designates the sharing of source code as a necessary freedom for all programmers, whether producers, consumers, or both. As such the source code must be “give[n]…away free,” Stallman writes, “to everyone who can use it.” Stallman did not mean that the commercial endeavors had no place in the software programming culture and later clarified this point in a subsequent e-mail “The GNU Manifesto.” In this post, Stallman states that his use of ‘free’ is meant to invoke “freedom” as in free speech, not “free beer”: “The important thing is that everyone that has a copy has the freedom to cooperate with others in using it.” As such, Stallman constructed code as the means by which to live and program by the golden rule; he writes, “I consider that the golden rule requires that if I like a program I must share it with other people who like it.” The black-boxing of software is thus constructed by Stallman as a practice that runs counter to morality. To act according to his will as well as to his morality means for Stallman not only the release but also the freeing of source code for everyone’s use and need. If everyone is given access to source code, which is programmer-readable, then everyone is free to share and share alike, thereby creating the free flow of the information demanded by a free society in which individuals are free to act according to their morality and will—the hallmark of Kantian idealism.

To do anything that prevents such freedom is for Stallman, as the author and narrator of this story, a violation of his “principles.” Having lived through the destruction of the community at the MIT AI Lab, which crumbled when programmers were hired away by

24 In a 1992 footnote added to the “GNU Manifesto,” which Stallman originally wrote in 1984 soon after his announcement of the GNU Project, he talks about his “careless” wording in his use of “free,” a carelessness that began with “new Unix implementation.” Stallman writes, “The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don’t make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent…Free software is software that users have the freedom to distribute and change. Some users obtain copies at no charge, while others pay to obtain copies—and if the funds help support improving the software, so much the better.”
proprietary software vendors\(^{25}\) and having previously suffered at the hands of proprietary software owners who had denied him access to source code that he needed to modify in order to work properly with his hardware,\(^{26}\) Stallman in no way wished to make others into victims by denying them the freedoms that had been denied him. Consequently, the story of a “new UNIX implementation” constructs the choice between proprietary software and free software, including the reactivation of practices associated with each, as a moral choice between creating victims through black-boxing and promoting freedom through transparent source code. To this end, Stallman rejects those practices that serve the black-boxing of code; Stallman writes, “I cannot in good conscience sign a nondisclosure agreement or a software license agreement.” To participate in the production of black-boxed software, which at the time typically involved the signing of such agreements, translated into an immoral choice for Stallman, who desired to live according to his will, which demands freedom.

To live the golden rule, to act morally, and to enjoy freedom, according to Stallman’s narrative means being able to share source code so that users could use, redistribute, and modify that code as they wished, thereby ensuring their inalienable right to act in way commiserate with their principles and morals. It is this relationship between developers and consumers rooted in a shared desire for a future steeped in the cultivation of democratic freedom inspired by Kantian-like idealism that Stallman believes should serve as the philosophical cornerstone and ideological trajectory for the software programming culture and a disruptive software technology.

\(^{25}\) Stallman recounts the death of the AI Lab as he knew it as follow: “It is painful for me to bring back the memories of this time. The people remaining at the lab were the professors, students, and non-hacker researchers, who did not know how to maintain the system, or the hardware, or want to know. Machines began to break and never be fixed; sometimes they just got thrown out. Needed changes in software could not be made. The non-hackers reacted to this by turning to commercial systems, bringing with them fascism and license agreements. I used to wander through the lab, through the rooms so empty at night where they used to be full and think, ‘Oh my poor AI lab! You are dying and I can’t save you.’ Everyone expected that if more hackers were trained, Symbolics [a proprietary software vendor] would hire them away, so it didn’t seem worth trying…” (as quoted in Levy, p. 425).

\(^{26}\) “I had already experienced being on the receiving end of a nondisclosure agreement, when someone refused to give me and the MIT AI Lab the source code for the control program for our printer. (The lack of certain features in this program made use of the printer extremely frustrating.) So I could not tell myself that nondisclosure agreements were innocent” (Stallman, *Open Sources: Voices from...* 1999).
Audience as Fellow Idealists

Unlike the audience constructed by Gates’ narrative—amateur hobbyists programmers, capitalistic consumers of code, and/or thieves who steal software—the audience constructed by Stallman’s narrative are fellow idealists, motivated primarily by their principles rather than their wallets. From the idealistically like-minded individuals on the Usenet newsgroup, net.unix-wizards, the site to which Stallman sent his e-mail, Stallman seeks contributions that would help further the GNU Project. Specifically, Stallman asks “for donations of machines and money” from computer manufacturers and adds that donating machines will also help to ensure “that GNU will run on them at an early date,” thus enabling idealists to use this software born out of a desire for democratic freedom as soon as possible. Additionally, Stallman asks for monetary donations. Donating money, the narrator explains, will allow idealistically-driven programmers to receive compensation for their labor and services:

If I get donations of money, I may be able to hire a few people full or part time. The salary won’t be high, but I’m looking for people for whom knowing they are helping humanity is as important as money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Stallman’s idealistic alternative to the dominance of entrepreneurialism in the software programming culture, though revolutionary, is not so utopian so as to expect that programmers will somehow be free of material conditions that necessitate the need to earn a living. Before the PC a custom among many hackers who shared source code, including Stallman, had been to charge for services involved with the distribution of the software (e.g., purchase of tapes, mailing costs, time it takes to transfer the code to a tape or disk). Reinstating fees-for-services as a morally-superior practice for the PC era in addition to monetary donations to the GNU Project would create opportunities for those programmers

---

27 In a story on the GNU Project published in 1999, Stallman tells of his decision to sell copies of the free GNU Emacs to those who could not download a copy from the GNU site: “So the question was, what would I say to them? I could have said, ‘Find a friend who is on the net and who will make a copy for you.’ Or I could have done what I did with the original PDP-10 Emacs: tell them, ‘Mail me a tape and a SASE, and I will mail it back with Emacs on it.’ But I had no job and I was looking for ways to make money from free software. So I announced that I would mail a tape to whoever wanted one, for a fee of $150. In this way, I started a free software distribution business, the precursor of the companies that today distribute entire Linux-based GNU systems” (Stallman, 1999, Open Sources: Voices).
dedicated to this idealistic cause to earn money through compensation for their labor and services rather than from the code itself. Thus, in his appeal for computers and monetary donations, Stallman appeals in this narrative to like-minded individuals from the hardware manufacturing culture who want to foster the spread of the idealistically-driven democratic freedom that is the catalyst behind the GNU Project.

In addition to his solicitation of hardware manufactures, Stallman asks “for donations of programs and work” from fellow ideologically-motivated programmers: “Individual programmers can contribute by writing a compatible duplicate of some Unix utility and giving it to me.” The GNU Project is thus constructed as a possible warehouse for UNIX-compatible software that programmers could not only contribute to but also draw from for their own use. As such, the audience for this particular solicitation is more narrowly constructed as hackers, who, because of their custom of sharing software, would most likely see the intrinsic value of what Stallman was proposing through his concept of “free” software. And, because of the hacker love of the technical—an motive that had nothing to do with locating economic profit in the source code itself—Stallman’s audience would also be interested in the technical challenge that this project offered as a possible means by which to accumulate social capital.

Consequently, Stallman constructs the tasks that lay before those who want to contribute software to an OS that is not only compatible with UNIX but that will be better than UNIX because of “improvements” made by those working on the GNU Project. In this way, the ideologically-motivated project is constructed to appeal to hacker motivations and interests:

To begin with, GNU will be a kernel plus all the utilities needed to write and run C programs: editor, shell, C compiler, linker, assembler, and a few other things. After this we will add a text formatter, a YACC, and Empire game, a spreadsheet, and hundreds of other things…We will make all improvements that are convenient, based on our experience with operating systems. In particular, we plan to have longer filenames, file version numbers, a crashproof file system, filename completion perhaps, terminal-independent display support…
Stallman represents the GNU Project as a complicated technical endeavor that will not just replace but will surpass the quality of UNIX, but that will take the cooperation and dedication of a number of idealistically-motivated programmers, who comprise the “we” in Stallman’s e-mail. The greater the number of hackers who participated by donating software and their labor to the GNU Project, the greater the likelihood that others could use, modify, and redistribute source code across a wide range of hardware and software configurations, thereby increasing the compatibility and quality of the code. Unlike Gates’ narrative wherein “we” represents Gates and Allen as entrepreneurs socially positioned against his audience of hobbyists, Stallman’s use of “we” socially positions himself in alignment with his audience of hackers and those idealists in computer manufacturing whom Stallman hopes will band together not for profit but instead for the production and sharing of the information that makes work of programmers possible. Like Gates’ narrative, Stallman’s narrative reifies the social positioning of those who share source code—typically referred to by the time of Stallman’s narrative as hackers rather than hobbyists—against entrepreneurs who black-box their source code and prevent the redistribution of software.

Because for Stallman it is social rather than economic capital that is valued most by hackers, Stallman, as the leader of the GNU Project, lays out his own hacker credentials for his fellow hackers by answering the question—“Who Am I?.” To this question, Stallman replies, “I am Richard Stallman, inventor of the original much-imitated EMACS editor…I have worked extensively on compilers, editors, debuggers, command interpreters, the Incompatible Timesharing System and the Lisp Machine operating system...” Because of the importance of social capital to hackers, Stallman identifies himself as a fellow hacker by reviewing the software contributions that he has made, many of which his audience would have been familiar with and also would have used on a regular basis, as was the case with the EMACS editor. Consequently, Stallman’s narrative construction can be understood as an appeal to hackers to unite with him in order to resurrect the principled traditions that have been criminalized because of the commodification of software through the entrepreneurial

---

28 In software programming, a text editor is a program used to write in programming language; in layperson terms, the function of the text editor is similar to those functions provided by a word processing program. *The New Hacker’s Dictionary* defines EMACS as “the plus ultra of hacker editors” (Raymond, 1996, p. 172). EMACS was originally written by Stallman in 1976. As part of the GNU Project, Stallman began developing GNU EMACS in 1984, and it was in wide use by 1985. The most recent version of GNU EMACS is 11.2, which was released in February 2005.
narrative. In doing so, Stallman and other hackers stand poised to follow the trajectory for future action mapped by this narrative, a trajectory rooted in golden-rule inspired morals rather than in fattening their wallets.

*Another Story of Good vs. Evil*

Like the entrepreneurial narrative of Gates, Stallman’s narrative “new UNIX implementation” constructs the theme of good vs. evil. But those on the side of good and those on the side of evil are inverted in Stallman’s construction, as might be expected in a narrative of resistance. On the side of good are those who live according to the moral principle of the golden rule that implicitly—and the hacker ethic that explicitly—advocates sharing “free” redistributable software and transparent source code with thy fellow programmers. Thus, in this narrative to live by the golden rule within the software programming culture is to share copies of software—the very practice that was constructed as stealing in Gates’ narrative—and, most importantly, to share transparent source code, something that is not even in the realm of possibility in the entrepreneurial story of software. Additionally, the sharing of code out of which all software—whether a tool, an OS, or an application—is constructed as a necessary freedom for living a moral life. To deny others access to source code is to act immorally and undemocratically by denying others access to information. In comparing the sharing of code production, which occurs through programming languages, to speech, the sharing of software thus becomes not only a moral obligation but a secular one as well just as with freedom of speech. By developing and sharing code according to these obligations, programmers uphold and spread a spirit of democratic cooperation within not only the software programming culture but also humanity as a whole, according to this narrative of Kantian-like idealism.

On the side of evil in this idealistic narrative are those who prevent the redistribution of software and black box their source code, thereby treating code as a commodity, the value of which lies in the profit it generates for its producers rather than in the service it can provide to programmers as well as to humanity. Unlike Gates’ narrative, which constructs this practice as one that best benefits the market, which in turn benefits consumers, which in turn benefits entrepreneurs, which in turn benefits society, Stallman’s story constructs the practice of black-boxing as both immoral and illegal because it goes against the golden rule
and the hacker ethic, both of which advocate the sharing of digital information. According to Stallman’s narrative, in a world of black-boxed non-redistributable software, producers and consumers are socially positioned against one another: producers want to maximize profit and consumers want to minimize cost. In a culture that shares software freely, producers and consumers are not socially positioned against one another but instead are united in and through their idealism, the consequence of which is a culture and society in harmony. Consequently, the desire of the producer and the desire of the consumer are constructed as one in the same.

By banding together and contributing to a project such as GNU, idealistic programmers and other like-minded individuals in the computer hardware industry can reclaim, according to the story constructed by Stallman, the moral, principled life embodied in the golden age of hackerism that existed in places like MIT’s AI Lab before the birth of the PC. Software programmers and users will not have to sacrifice their principles by creating and using source code that they are not allowed to share but instead will be able to share and share alike as their morals dictate. However, without contributions to and support for idealistically-motivated causes such as the GNU Project, Stallman argues that the future of the software programming culture will continue to be mired in the immoral practices prescribed through the entrepreneurial narrative. For it is not only on the hacker programmers that the commodification of software takes its toll but also on all idealistically-motivated individuals who believe in a Kantian-like democratic freedom in which the ability to act morally is valued more than monetary gain.

With the entrenchment of the PC software programming culture in the entrepreneurial narrative of Gates and the subsequent rush to profit economically through the sale of established software technology as a commodity, Stallman rhetoric is revolutionary. In constructing an ideological trajectory for future action, Stallman argues that this trajectory should privilege democratic morality over entrepreneurial capitalism. As such, Stallman wishes to resurrect a trajectory that existed before the entrepreneurialism of PC software and, in doing so, develop disruptive software technology. Through a software renaissance, programmers would once again be free to share and have significant opportunities to do so. In short, the narrative construction of “new Unix implementation” calls for a revolution of
morality to stop the revolution of capitalism that had begun just seven years before in the PC software programming culture.

**The Pragmatic Narrative**

Unlike the entrepreneurialism espoused by Smith or the transcendental idealism advocated by Kant, pragmatism, especially as developed by John Dewey, emphasizes the development of skills, objects, mechanisms, and means for use in the solving of practical problems that arise out of experience. According to Dewey, the need for practical means in solving problems can only be identified by an individual’s personal interest and exploration in cooperation with others rather than by some overarching philosophy or dogma. Thus, dogmatic philosophies, such as those associated with the materialism of capitalism or the morality of idealism, problematically identify an abstract idea as the end from which the means stem. What Dewey advocates instead is allowing the means for solving a problem to develop democratically out of practical experience and interest rather than from dogma. In this way, the end arises out of those means that worked best in problematic situations.

For example, Dewey argues that as a result of the kind of democracy born of the dogma of “finance capitalism” (1998a, p. 337) individuals are controlled, sapped of their freedom by “dominant corporateness” (1984, p. 18). This corporateness “determines the tone of society at large as well as the government of industrial society” (p. 21). Consequently, individuals do not participate democratically in the “development of a shared culture” but instead are born into forces that prescribe already established means with the end result being a democracy that serves capitalism at the expense of individuals (p.17). As a result, Dewey argues “our indigenous heroes are the Fords and Edisons who typify this mind to the public” (p. 21).

---

29 In Democracy and Education, Dewey connects play with work, two activities that traditional U.S. education constructed through the dogma of capitalism treats as mutually exclusive. Dewey contends that the pragmatic approach identifies work as extended play (1916, pp. 203-205). As children, play reflects an individual’s interests. “Persons who play are not just doing something…they are trying to do or effect something, an attitude that involves anticipatory forecasts which stimulate their present responses. The anticipated result, however, is rather a subsequent action than the production of a specific change in things. Consequently play is free, plastic…If a child is making a toy boat, s/he must hold on to a single end and direct a considerable number of acts by that one idea” (p. 203). Work for the pragmatist is simply a matter of a more complicated and lengthy process of activity, consequently, “Work which remains permeated with the play attitude is art—in quality if not in conventional designation” (p. 206).
Like capitalism, the dogma of idealism does not begin with experience but with a
dogmatic abstraction, according to Dewey: “the idea that morals ought to be, even if it is not,
the supreme regulator of social affairs…is still advanced by sermons from the pulpit and
editorials from the press that adoption say of the Golden Rule would speedily do away with
all social discord and trouble” (1989, p. 19). By cultivating pragmatic principles, rather than
being inscribed by dogmatic principles, through the development of tools for problematic
situations, individuals increase their potential to create knowledge that serves society.

According to Dewey’s pragmatist school of thought, it is only by directing individual
attention to experience-driven “problems of interest” that knowledge is cultivated (1964, p.
200). Accepting that means of action originate from dogmatic philosophies rather than
specific experiences closes off avenues by which to develop and test practical solutions. The
limits imposed by dogmatic thinking, which typically results in dualisms, originate from
thinking of the end as a solution, fiat accompli, and the existing means associated with that
end as always having been and always continuing to be the means for achieving said end.
Deweyian pragmatism, on the other hand, begins with the means; the shape that the end takes
is consequently dependent on those means developed out of practical experience (1989, p.
134). To put the ends before the means, as dogmatic thinking does, prevents, according to
Dewey, the development of practical skills and objects—“things to be put to use, and that use
is their contribution to a common and shared life” (p. 11).

This is not to say that pragmatists do not themselves advocate a particular end.
Dewey contends that the pragmatist end to be achieved is democracy. However, the means
by which to achieve this democratic end are not readily determined by some transcendental
idea inscribed by dogma. Only through the continual wrestling with the everyday existence
and problems of humanity that arise out of existence are the means to this end as well as the
manifestation of the end itself revealed. As Dewey states, “democracy can be served only by
the slow day by day adoption and contagious diffusion in every phases of our common life of
methods that are identical with the ends to be reached and that recourse to monistic,
wholesale, absolutist procedures is a betrayal of human freedom no matter in what guise it
presents itself” (p. 133). To this end, Dewey advocates the scientific method, which
encourages the creation, testing, and adoption of knowledge through consensus. To achieve
consensus, scientists, who rely upon the findings of other scientists, must freely share their activities with other members of the scientific community. Through the democratic “habits” of cooperation and participation, scientists solve practical problems, stand poised to garner social capital if their work is accepted, and contribute to the knowledge of the community. Adhering to the scientific method in the pursuit of democracy allows such an aim “which, although ideal, [is] not located in the clouds but [is] backed by something deep and indestructible in the needs and demands of humankind” (p. 120).

But the scientific method demands freedom, the ability “to make one experience freely available in other experiences” (Dewey, 1964, p. 339). Such freedom benefits both the individual and the community in their mutual cultivation of knowledge. Because the circumstances in which experience occurs are never replicable, the practical use of objects demands continual development, testing and modification to meet those changing circumstances. The optimal way in which to do so, according to Deweyian pragmatism, is through cooperation and participation with others so that “we can face our problems in detail one by one as they arise, with all the resources provided by collective intelligence operating in cooperative action” (1989, p. 134). It is through these elements of pragmatism—the blurring of the boundary between work and play as well as the importance of participation and cooperation in the collective creation of knowledge to meet practical needs and problems—that Linus Torvalds sets a pragmatic trajectory of future action for the software programming culture through his story of software.

In August of 1991, 15 years after Gates’ “An Open Letter to Hobbyists” and eight years after Stallman’s “new Unix implementation, Torvalds, a graduate student in computer science at the University of Helsinki, posted “What would you like to see most in minix?” to the MINIX newsgroup MINIX-L. In this e-mail, Torvalds announced his work on a MINIX-like OS—what would eventually become the Linux kernel, the heart of the

---

30 Stephen Fishman (1993), in his exploration of Dewey’s conception of community as it relates to the composition classroom, emphasizes the importance of social capital to the scientific community; “Scientists, unlike entrepreneurs, cannot claim individual ownership of their products because, to win respect, their theories and findings must be shared and tested by their peers” (p. 318).

31 MINIX, originally developed by Andrew Tanebaum in 1987, is a UNIX-like OS. According to its creator, “MINIX has been written from scratch, and therefore does not contain any AT&T code—not in the kernel, the compiler, the utilities, or the libraries. For this reason the complete source can be made available (by FTP or via the WWW)” (Tanebaum). Like UNIX, the trademark for this OS is capitalized. Consequently, I use “MINIX” unless quoting from someone who writes “Minix.”
GNU/Linux OS: “I’m doing a (free) operating system (just a hobby, won’t be big and professional like gnu).” Additionally, Torvalds solicited feedback about what his readers liked and did not like about the proprietary OS MINIX. In October of the same year, Torvalds sent a follow-up e-mail, “Free minix-like kernel sources for 386-AT,” in which he constructs his story of Linux. Torvalds’ narrative begins with the announcement to the newsgroup that the MINIX-like kernel is “usable” though still in a very early stage of development, continues with the solicitation for others to use the source code, and concludes with the request that other programmers to share whatever “free” software they have that might advance the development of the system. Through his projectivity, Torvalds constructs in this narrative a trajectory of future action for the software programming culture that rejects the dogmas of both entrepreneurialism and idealism and that instead emphasizes the solving of practical programming problems not for profit or for morality but for interest and enjoyment. Thus, Torvalds’ narrative socially positions himself and programmers like him outside of the adversarial set of relations constructed through the narratives of Gates and Stallman, and instead creates a social positioning that is sometimes aligned with and sometimes aligned against both of these trajectories, as the following analysis illustrates.

**Narrator as Pragmatic Hacker, Problem-Solver, and Scientist**

Torvalds’ narrative, “Free minix-like kernel sources for 386-AT,” reflects a pragmatic approach to software development, and as such, Torvalds, as the narrator, constructs himself as a pragmatist. First, Torvalds identifies the Linux kernel as a “pet project of mine.” Though a disruptive software technology, Linux is neither an entrepreneurial enterprise nor a moral cause but rather a project begun because of a practical programming problem (Torvalds was unable to connect his home computer with his university’s computer). And, even though the

---

32 In this e-mail, Torvalds refers explicitly only once to the kernel he is working on as “linux,” which had yet to become the official name. However, he does implicitly refer to Linux when listing the ftp directory to download the source code, which was “/pub/OS/Linux.” In Just for Fun: The Story of an Accidental Revolutionary, Torvalds reveals his uneasiness with the name “Linux,” which was meant to be a private working name. The decision to make “Linux” the official name was Ari Lemke’s, a teaching assistant at the University of Helsinki. Torvalds write, “I admit that I didn’t put up much of a fight. But it was [Lemke’s] doing. So I can honestly say I wasn’t egotistical, or half-honestly say I wasn’t egotistical. But I thought, okay, that’s a good name, and I can always blame somebody else for it, which I’m doing now” (Diamond & Torvalds, p. 88).

33 Torvalds never specifies in his e-mail the practical need that prompted his work on Linux. In Just for Fun: The Story of a Accidental Revolutionary, published 10 years after the e-mail “Free minix-like kernel sources for 386-AT,” Torvalds writes that his work was prompted by the inability to connect using Minix his personal computer with the university’s computer.
kernel for Stallman’s GNU project OS, was in development and would be available at some point, as Torvalds acknowledges in his narrative—“Hurd will be out in a year or two, or next month, who knows”—he continued developing the Linux kernel because it solved practical problems that he as a programmer was experiencing. Additionally, the development of Linux is something undertaken by Torvalds out of personal enjoyment—“I’ve enjoyed [sic] doing it…,” he writes. Linux provides the means for his own enjoyment, the kind of enjoyment that every technically-oriented nerd/geek, especially a hacker, desires. Torvalds constructs himself as hacker, though neither the criminal hobbyist constructed by Gates nor the idealist hacker constructed by Stallman. Neither does Torvalds construct software in general as having no intrinsic economic value of its own and thus shareable; nor does he construct Linux specifically as an anti-entrepreneurial idealistically-motivated alternative to proprietary MINIX. Torvalds instead locates the impetus for his own software development not in profit or morality but in a problem and interest that arose out of his own experience—to make his computer function as he needs it to and to indulge those interests that bring him enjoyment.

Consequently, according to his narrative construction, Torvalds does not create his “free version of a minix-lookalike” to make a profit or to live according to his morals but to use his skills to develop the software to solve a programming problem as well as to share freely the code he creates with others who have a problem that his code might help to solve. Like Stallman, Torvalds sees a benefit to sharing software, but not for the idealistic reasons Stallman does. The freedom to redistribute software and to share source code that can be modified allows other programmers to use the software Torvalds develops to address their own programming needs. This absence of idealism and its necessary condemnation of the black-boxing of non-redistributable proprietary software, as well as the absence of praise for sharing as a practice stemming from the Golden Rule, differentiates Torvalds from Stallman, especially given that some of the library sources that Torvalds uses in conjunction with his system so that he could check his e-mail at home. Having accomplished this task through the code he developed, Torvalds then turned to developing code that would allow him to download and upload files at home. This process of fulfilling practical needs continued until Torvalds recognized that “the project was on its way to becoming an operating system” (Diamond & Torvalds, pp. 62-78). Upon reflection, Torvalds writes, “My original goal was to create an operating system that I could eventually use as a replacement for Minix. It didn’t have to do more than Minix, but it had to do the things in Minix that I cared about, and some other things that I cared about, too” (p. 81).
development of Linux are black-boxed. Torvalds writes, “Full kernel source is provided, as no minix code has been used. Library sources are only partially free, so that cannot be distributed currently.” Of primary importance for Torvalds is accomplishing his task of making his computer work the way that he needs it to, not issues of profit or morality. Thus, what occupies Torvalds is the solution to his programming needs, needs that other programmers might share. If only black-boxed non-redistributable source code is available to help him in his endeavor, then Torvalds is willing to use that software. However, given that Torvalds, who constructs himself as a practically-oriented hacker, makes Linux free rather than transparent but proprietary (as with UNIX) or proprietary and black-boxed (as with most Microsoft BASIC), the freedom to share software with transparent source code is thus constructed as the most useful means by which to solve his programming problems. As such, the transparency of the free Linux kernel source code does not share the same ideological encoding as the transparency of the free GNU Project OS and software, which is a result of moral idealism.

As the narrator, Torvalds constructs himself as a provider of free code, a solicitor of free code, and a user of free code developed by other programmers. By uploading Linux to server space from which others can then download it, Torvalds made both the software and the source code available to anyone with Internet access: “I am willing to put out the sources for wider distribution…Sources…can be found at nic.funet.fi (128.214.6.100) in the directory /pub/OS/Linux.” But Torvalds, as the narrator, is not constructed solely as a provider of code, as Gates is, but also as a solicitor of code. Torvalds writes, “I’m also interested in hearing from anybody who has written any of the utilities/library functions for minix. If your efforts are freely distributable…I’d like to hear from you, so I can add them to the system.” Torvalds welcomes software from anyone as long as the source code is available for everyone’s use, including his own, and redistributable. Thus, Linux is constructed as a project that would benefit from the abilities of all who contribute to it. To demonstrate the usefulness as well as the social capital that can be accrued by working on Linux, Torvalds

---

34 Even when Linux had begun to receive interest from outside the software programming culture, Torvalds had no qualms using black-boxed software if that allowed him to accomplish his task. This pragmatic approach explains why in 1998 Torvalds for his address evangelizing Linux at Comdex, the world’s largest computer-related conference, used Microsoft PowerPoint. Though black-boxed and non-redistributable, PowerPoint best served his needs.
sends a compliment via email to a programmer who has already contributed code that has been added to Linux: “I’m using Earl Chews estdio right now (thanks for a nice and working system Earl)….” The exchange of free software is constructed as a mutually beneficial relationship that programmers can have with one another through the sharing of redistributable source code. Through the work of many programmers working in accord, and adopting the democratic habits of cooperation and participation, to address their own programming needs, the Linux kernel stands to grow and improve, thereby benefiting everyone who uses it, including Torvalds.

Linux’s use at the time that Torvalds wrote “Free minix-like kernel sources for 386-AT” was extremely narrow because Torvalds was the only programmer, Linux was only in “version 0.02”, and because of the constraints of Torvalds’ own hardware and software configurations. Torvalds was using an AT-386 computer and the MINIX OS for the 386; consequently, Linux was coded to work on that system and that system only. Torvalds writes in his e-mail, “ALERT! WARNING! NOTE! These sources still need minix-386 to be compiled (and gcc-1.40, possibly 1.37.1, haven’t tested), and you need minix to set it up if you want to run it, so it is not yet a standalone [sic] system for those of you without minix. I’m working on it.” By freeing Linux’s source code, Torvalds opens up the opportunity for other programmers to test his code, to make modifications to his code, and to redistribute freely those modifications so that Linux develops or, should experience necessitate it, cease to develop based on the needs of every programmer in the group.

Through this scientific method of programming (i.e., the sharing of one’s work with others so that they can test, modify, and redistribute the code with or without modifications), the likelihood of increasing Linux’s quality (e.g., performance, reliability, compatibility, security, innovation) grows as the number of programmers, each with his or her particular hardware and software configurations, work on and contribute to the kernel. Thus, the development of Linux is dependent upon the scientific method (in Dewey’s sense of it) in that the creation of knowledge occurs through consensus, which can only take place through

35 Software programs are assigned numbers to denote the version of the code. The higher the number, the more seamless and advanced the features. Version 1.0 signifies that a program is ready for the public, though the use of the program determines whether or not the “public” is technical or lay. Torvalds’ release of Linux 0.02 represents the earliest stage of development. 0.01 is the only version more rudimentary; Torvalds had released Linux 0.01 a month before in September 1991. As of 2005, Linux is at version...
freedom. So while Linux’s use was still very limited at the time of the narrative, something Torvalds was quick to point out, he constructs this limitation as part of a natural process of experimentation in code development. As such, Torvalds, as the narrator, constructs himself as the lead scientist undertaking a new project, which he hopes others will participate in and, in doing so, form a scientific community whose work benefits the practical programming needs of all involved.

**Audience as Fellow Pragmatic Hackers, Problem-Solvers, and Scientists**

Unlike Gates, who constructs himself in opposition to his hobbyist audience, and like Stallman, who constructs himself in alignment with his audience of fellow idealists, Torvalds’ audience construction mirrors that of his own construction as the narrator—his audience includes possible providers, solicitors, and users of free redistributable and modifiable source code, who together form a scientific community. Because Linux is a project “for hackers by a hacker,” Torvalds’ audience is constructed as having the same practically-oriented needs for source code as he has—to indulge those interests that bring enjoyment and to have their computers function as they need them to. Torvalds writes, “…Somebody might enjoy looking at [Linux’s source code] and even modifying it for their own needs.” Thus, Torvalds’ audience is very much like Torvalds himself—interested in solving practical software problems that arise out of their programming experiences and enjoying the process of doing so. In fact, Torvalds’ opening lines of his e-mail to the MINIX newsgroup point to the need for such enjoyment, something he believes programmers can have through Linux: “Do you pine for the nice days of minix -1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a [sic] OS you can try to modify for your needs? Are you finding it frustrating when everything works on minix? No more all-nighters to get a nifty program working? Then this post might be just for you :-).” Part of the enjoyment in working on software, especially in its earliest stages, is the social capital that programmers can accrue through their work, as this quote evidences. In the software programming culture, a measure of nerd/geek prowess is the degree of difficulty in creating a piece of code. Writing an OS from scratch is considered the most difficult of coding enterprises; consequently, Linux offers a number of opportunities for
garnering social capital. In fact, even to set up Linux 0.02 demands sophisticated technical abilities, as Torvalds warns; “You...need to be something of a hacker to set [the kernel] up.”

But because Torvalds is concerned with the practical, he does further narrow his immediate audience to those who have a certain hardware and software configuration that allows Linux to run on their computers; “ALERT! WARNING! NOTE!” draws attention in his e-mail to the technical specifications that are need to run Linux successfully. To emphasize the limited compatibility of Linux, Torvalds first explains whom Linux is not for—“for those hoping for an alternative to minix-386, please ignore me,” and then continues with an explanation of his target audience: “It is currently meant for hackers interested in operating systems and 386’s with access to minix.” Although Torvalds’ appeal to hacker sensibilities in the opening paragraph would certainly appeal to hackers beyond AT-386 users, the possibility of Linux running on any other computer system does not yet exist; thus, the number of programmers participating in this scientific community focused on Linux development is limited. By banding together to use and test Linux, provide feedback on its performance, and contribute code to the kernel, hackers can extend the community to include those hackers with other system configurations, but if and only if they are interested in doing so and have a practical need for what the code allows a computer to do. With a community of hackers cooperating with one another, using the scientific method, and participating in the creation of knowledge in the continued development of the Linux kernel, the possibility exists to increase Linux’s usefulness not only for Torvalds but for all who use it.

Beyond Stories of Good and Evil

The narrative construction of “Free minix-like kernel sources for 386-AT” avoids both the capitalistic dogma of entrepreneurialism perpetuated by Gates’ story and the idealistic dogma of morality forwarded by Stallman’s. Also absent is the binary of good and evil. Sharing is not evil because it is criminal; nor is sharing good because it is moral. Instead, sharing is constructed as the most feasible way to create quality software that meets the needs of its users because the practice allows the greatest number of people to participate in the creation, testing, and modification of the source code. Rather than being concerned with monetary profit or moral principles, Torvalds’ interest lies in the practical endeavor of developing code for specific situations. This endeavor is not cultivated by some abstraction
but by Torvalds’ enjoyment and interest in writing code that fulfills a practical programming need that other software does not.

By focusing on the development of code to solve a problem that arises out of his experience, Torvalds provides an alternative to the dominant discursive coding of software as a for-profit commodity and the potentially-revolutionary discursive coding of software as a moral endeavor. By placing the means for completing a task or solving a problem first, Torvalds resists the dogmatically-specified abstract end that situates code creation in the service of some higher purpose, whether economic profit or the golden rule. Consequently, cooperation and participation are practiced not because they are inherent democratic freedoms but because the quality of software written for a specific purpose can be increased with the increase in the number of programmers using, testing, and modifying the code. However, a consequence of these acts is democracy because cooperation and participation in the development of software is proven to be the best way to solve the practical problems of programming. Thus, the end that lies beyond a specific situation is of little consequence for Torvalds as a pragmatist. The end created out of practically-oriented means that have proven useful in everyday life could lead to end that is similar to that of entrepreneurialism or idealism. Or the end may be a hybrid of the two or may take shape as something completely different than either. But whatever the shape this end takes does not occupy Torvalds because in the end what matters is whether or not the best tools are available to do the best job.

The earliest archived narratives of Gates, Stallman, and Torvalds set always competing, though not always antithetical, trajectories of action for the software programming culture, while at the same time reifying both the culture itself and the concept of democratic freedom. Through Gates’ entrepreneurial narrative “An Open Letter to Hobbyists,” he challenges the hobbyist construction of software as shareable and instead imagines, through his projectivity, a future dedicated to the economics of a free market that construct agency as the freedom to produce, to consume, and to generate wealth. As such, software is an intellectual property that exists only as a non-redistributable commodity with black-boxed source code that prevents others from sharing the program, reading the software in programming language, and modifying the code. For any individual to redistribute copies of software is a crime against not only the software vendor but against both law-abiding
citizens and the nation as a whole. The former is deprived of innovative products because the vendor is not able to recoup the largest amount of capital from the sale of the use of its software, a process that affects the vendor’s research and development expenditures; the latter is deprived of revenue that grows the national economy.

In contrast to this capitalistic trajectory, Stallman’s idealistic narrative “new UNIX implementation” constructs a very different future for the software programming culture, one that destabilizes the construction of software as intellectual property to be produced and consumed as a commodity. Through his projectivity, Stallman imagines for the software programming culture a future in which morality takes precedence over profit. To act morally as a software developer is to follow the golden rule in the tradition of the hacker ethic. Only by freely sharing copies of redistributable software with transparent and modifiable source code—in other words, free software—with their fellow programmers can developers live a principled life. To not allow such a practice denies all who develop and use software the freedom to act according to their will and thus denies actors agency. The consequence of this denial is the denial of democracy freedom, which only exists insofar as individuals are able through their agency to act on their principles unimpeded, which in the software programming culture translates for idealists such as Stallman into free software.

Though the entrepreneurial and idealistic narratives are antithetical both ideologically and practically, Torvalds’ pragmatic narrative has differences and similarities to both. Torvalds through his projectivity imagines for the software programming culture a pragmatic trajectory for future action in which programmers develop, use, test, modify, and redistribute software both because they enjoy it and because they have a practical need to do so. Thus, this pragmatic construction of software rejects the dogmatism of both the entrepreneurial and idealistic narratives but not in a dogmatic way. Torvalds, for instance, does not condemn proprietary software; if non-redistributable black-boxed source code is what works best for his needs, he uses it. If, on the other hand, shared source code is available and meets his needs, he uses that. When developing software himself, what works best for Torvalds (what in his experience works best for the creation of quality software) is to develop free software that he can share with others so that they too can cooperatively use, test, modify, and redistribute that code. Thus, to have agency according to the pragmatic narrative of software
is to be able to use the best tools available in the solving of practical programming problems. As such, the sharing of free software is encouraged but not because it is a morally superior practice or because it generates monetary profit but because the larger the number of programmers using, testing, and modifying the source code the better the software.

Though Gates, Stallman, and Torvalds construct through their respective narratives competing trajectories for the software programming culture, none reject the culture or its primary occupation—the development of software. In fact, these narratives clearly construct the significance of the culture and its iterative practices as worthy of struggle. As such, these narratives, not in spite of but rather because of their differences, are unified in their mutual reification of the culture and its set of shared cultural practices (Figure 4). In their evangelization of particular paths of software development, Gates, Stallman, and Torvalds reproduce not only the software programming culture’s iterative practice of evangelism but also the practices of programming, instrumentalist discourse about quality, and the regulation of code. Thus, we can understand these stories of software as both an effect of and effect upon these practices, especially in terms of their significance. These iterative practices that I identified and described in the previous chapter are reactivated, according to Emirbayer and Mische, in “largely unreflective and taken-for-granted” activities (Emirbayer & Mische, p. 973). By now directing attention to the projectivity of actors, which involves “a first step toward reflectivity” (p. 984), we can understand the ways in which programmers such as Gates, Stallman, and Torvalds ideologically encode software as well as of those who develop and use particular kinds of code.

**THE EVOLUTION OF THE COMPETING IDEOLOGICAL TRAJECTORIES**

Though there is certainly danger in attributing to any one moment of storytelling the significance that I have thus far attributed to the respective narratives of Gates, Stallman, and Torvalds, the representations constructed in these stories of software are in no way limited to the specific narratives of “An Open Letter to Hobbyists,” “new UNIX implementation,” and “Free minix-like kernel sources for 386-AT.” The meaning of software and the software programming culture constructed through the earliest archived narratives of Gates in his construction of software as a commodity, Stallman in his construction of software as morality, and Torvalds in his construction of software as a tool have been reproduced and
refined over and over again in and through the evangelism of these then yet-to-be spokespersons. Reaffirmed in the evolution of these narrators’ respective evangelism is software as a commodity, software as morality, and software as a tool. I offer examples of the evolution of these ideological encodings in order to illustrate how the narrative constructions first offered in 1976 (Gates’ “An Open Letter to Hobbyists), 1983 (Stallman’s “new Unix implementation”), and 1991 (Torvalds’ “Free minx-like kernel sources for 386-AT”) have rhetorically developed.

Software as a Commodity: Gates’ Evangelism

In his continued evangelism of the entrepreneurial trajectory for software and the software programming culture, Gates has insisted, “everything we [Microsoft] believe in terms of the role of software, the degree that it’s going to get better, hiring smart people, our basic principles are the same today as they were when the company was founded” (October 1, 2004). As such, Gates has symbolically worked to stabilize the seemingly natural connection between software, market capitalism, and the democratic freedoms to produce, to consume, and to generate wealth: “Capitalism, demonstrably the greatest of the constructed economic systems, has in the past decade [of proprietary software development] clearly proved its advantages over alternative systems” (1996, p. 207). The evidence of this advantage is located in the wealth generated by entrepreneurs such as Gates and Allen, whose personal wealth in 2004 was estimated at $46.5 billion and $21 billion, respectively (Kroll & Goldman). To situate software and its developers in a system that allows for the possibility of such an outstanding accumulation of personal wealth has been for Gates an unqualified good, but not the only one. Another unqualified good has been the development of a new industry that grows national economies and provides employment opportunities for individuals; after all, “There essentially was no software industry before the PC,” Gates notes (2004). Through their kairotic recognition of the software frontier that laid before them in 1975 and their adventurous entrepreneurial spirit, Gates and Allen, according to Gates’ recurrent narrative, envisioned a future that they made real through their actions:

We knew that getting a company started would mean sacrifice. But we also realized that we had to do it then or forever lose the opportunity to make it in microcomputer software…We thought we saw what lay beyond the Intel 8080 chip and then acted on
it. We asked, “What if computing were nearly [italics added] free?”…We set up shop betting on cheaper computer power and producing software when nobody else was. Our initial insight made everything else easier. We were in the right place at the right time. We got there first, and our early success gave us the chance to hire more and more smart people. We built a worldwide sales force and use the revenue it generated to fund new products. But from the beginning, we set off down a road that was headed in the right direction. (1996, pp. 19-20)

As Gates recounts his own temporal orientation to the future at the earliest beginnings of the PC era, we can identify the consistency between his entrepreneurial construction of software as a commodity in his “An Open Letter to Hobbyists” and his recreation 21 years later of the exigency that led to the development of Altair BASIC. “To create a thriving personal computer and software business,” the billionaire Gates writes, “that was our dream” (February 26, 2004).

However, in order for the capitalist system and the entrepreneurial construction of software to continue to benefit producers and consumers as well as the nation, Gates warns, “it’s the investment in research, the value of intellectual property…that we have to renew our commitment to” (February 24, 2004). Doing so allows “a very large part, dramatically larger than ever before, of the world’s population…to participate in capitalistic opportunity” (October 1, 2004). But it is not only the citizens of the world that benefit from a capitalistic approach to software. Just as Smith theorized, the nation also benefits but only through the entrepreneurial construction of software: “the U.S., “Gates argues, “has to keep its edge in terms of doing the best work and that means research, it means intellectual property” (February 26, 2001). Because, as Gates acknowledges, Linux “has been gaining” (February 24, 2001) in the UNIX business enterprise market, the freedoms born of the economic market are possibly threatened, and democracy itself is under attack.

For instance, for governments that develop software in-house to consider because of the success of Linux what Gates considers to be a non-capitalistic approach to software is particularly problematic not only for him but also for the nation in that the government is renouncing its capitalistic freedom as well as its duty to create employment opportunities and to generate wealth for the nation; Gates writes, “one thing that we get people [from the
government] discussing with us a lot is how to create jobs around [information technology] activity. And I think you will see some countries who really believe in the capitalistic approach; that is, that software should generate jobs, and government R&D should generate jobs, so that government R&D should be done on a basis that it can be commercialized” (April 17, 2002). Intellectual property, such as software, allows for the production of a commodity without material resources, and that consequently generates endless opportunity for profit for producers and for affordability for consumers; “the magic of intellectual property is you can invent the idea once and whether it’s a song or a drug or a piece of software, that can be made available to millions of people at extremely low cost” (2001). So for the U.S., for example, not to support proprietary software is detrimental to producers, consumers, and capitalistic enterprise, which given the entwinement of the American democracy with capitalism is particularly problematic. Likewise, software released with its source code is not something that most consumers want, according to Gates: “A lot of customers in a sense don’t want—the notion that they would go in and tinker with source code, that’s the opposite of what’s supposed to go on. We’re supposed to give that to them and it’s our problem to make sure that it works perfectly and they rely on us” (October 1, 2004). Consequently, Gates believes that “in terms of free software…[its] role…in the commercial environment will actually be fairly limited, but that is not to say that we do not think of it as competition” (March 26, 1999), a fact that he reiterated during Microsoft’s antitrust trial when Gates used Linux to illustrate how Microsoft is not a monopoly (Testimony). But more, recently, Gates has acknowledged, “the interplay [between free and commercial software] that’s working very well” (October 1, 2004).

In addition to his continued espousal of a capitalistic trajectory in commodity production and consumption of software, Gates also has continued his vilification of those whom in 1976 he called hobbyists and whom today he typically refers to as hackers. Though Gates does not directly construct those involved in the development of GNU/Linux as criminals, unless they act illegally, he has used the term ‘hacker’—which so many involved in the development of free software use positively to refer to themselves—to describe programmers involved in computer-related illegal activities. Speaking to future issues of concern involving software and the Internet, Gates has asked, “As our economy becomes
more dependent on bits than on atoms, how will we protect these resources from being
damaged or devalued by hackers?” (2000). Hackers, therefore, continue to be a threat, but not
just to proprietary software developers. Because information technology serves as the
infrastructure for much of modern day life, hackers are a threat not only to the generation of
wealth, which according to both Smith and Gates affects everyone, but also to daily living.
So although information technology is a “powerful tool to defend and secure the nations of
the world which guarantee basic freedoms,” Gates writes, “the fact that most cybercrimes are
never solved presents an open invitation to hackers, identity thieves and spammers who are
scammers” (2000). Consequently, hackers, who were commonly referred to in the 1970s as
hobbyists, continue to be constructed as criminals by Gates, but the extent of their
detrimental effect on producers, consumers, and the wealth of nations has grown with the
growth of the Internet and our increasingly digitalized lives. Just as Gates has continued to
criminalize hackers, he has also continued to vilify the practice of sharing software, what he
more commonly refers to as piracy.

Piracy, the “stealing” of proprietary software, is now just one of the crimes associated
with hackers or the activity of hacking, which Gates has used to denote a number of
computer-related crimes. As he explains, “The software industry has struggled with piracy
since the advent of the personal computer, but as recent controversy over file-sharing systems
such as Napster and Gnutella demonstrates, piracy is now a serious issue for any individual
or business that wants to be compensated for the works they create” (Gates, 2000). But with
the ubiquity of the PC, piracy is no longer limited to technically-proficient hobbyists or
hackers; anyone who uses a computer is now a possible pirate. And software is no longer the
sole object of piracy; digital content (e.g., DVD movies, books, music) and secure computer
systems can also be pirated. In spite of the increased extent to which hackers and pirates can
undercut the economic worth of proprietary commodities, the primary significance of piracy
continues to be the negative effect it has on the market due to the loss of capital accumulation
from the commodity of software. But with the prosecution of pirates, capitalism is reified,
according to Gates: “And so as piracy goes down…the market grows” (Frost & Gates).
Another benefit to the prosecution of criminal pirates who steal their software is the
recoupment of previously lost capital, which can then be reinvested not only in R&D but also
in helping poorer countries where piracy rather than purchase is the modus vivendi; Gates writes,

> One of the things we’ve done is whenever you do have, which you have in a lot of these countries, piracy actions we take all the money that’s recovered from that and put that back into…training programs for people who are out of jobs, and training of teachers. So, we’ve dedicated all of that to piracy recovery, which, over the last five years [from approximately 1995 to 2000] has been over $50 million that’s gone into the various retraining programs. (October 18, 2000)

Just as the policing of hobbyists as constructed in the “An Open Letter to Hobbyists” had benefits so to does the policing of software users in order to route out pirates; Gates writes, “We’re committed to working with governments to find hackers” (June 25, 2003). From Gates’ subsequent discourse, we are able to identify the discursive evolution of his story in “An Open Letter to Hobbyists” and the resulting stability that has arisen from Gates’ construction of software as a market commodity, of hackers as criminals rather than hobbyist technophiles, and of the positive effects of prosecuting those who deny others the freedom endowed by the capitalistic market.

**Software as Morality: Stallman’s Evangelism**

Like Gates, Stallman, who currently earns a living from awards and speech-making fees, has continued to reproduce the trajectory for future action that he set for the software programming culture in his 1983 narrative “new UNIX implementation.” Recounting his experience at the MIT AI Lab, the catalyst for his decision to create the GNU Project, Stallman has stated that he was left with “a stark moral choice”: to use and develop black-boxed proprietary software, thereby “making the world a worse place,” or “to leave the computer field” (2002b, p. 17). In order to prevent other programmers from facing a similar choice, Stallman has continued to reify the connection between software and morality. “My work on free software,” Stallman has written, “is motivated by an idealistic goal: spreading freedom and cooperation. I want to encourage free software to spread, replacing proprietary software that forbids cooperation, and thus make our society better” (2002a). To this end, Stallman has subsequently sought to alleviate some of the confusion surrounding his use of “free,” especially its problematic connotation of price or lack thereof. Specifically, Stallman
has created the “Free Software Definition,” in order to illustrate the ways in which “free” is akin to “‘free speech’… not… ‘free beer’” (1996). This definition is composed of four freedoms regarding software; the “Free Software Definition” reads as follows:

- Freedom 0: The freedom to run the program, for any purpose.
- Freedom 1: The freedom to study how the program works, and adapt it to your needs…
- Freedom 2: The freedom to redistribute copies so you can help your neighbor.
- Freedom 3: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits…

Just as free speech only exists inasmuch as there are no limits to it—for it is often only when those limits are put in place that “free speech” is invoked--so to with free software. According to Stallman, these four freedoms of software use, examination, modification, and redistribution allow both developers and users of free software such as the GNU Project to live a morally-driven life by not limiting the rights of other in the assertion of one’s own rights. After all, to charge for the limited use of non-re redistributable black-boxed software is like charging for air, according to Stallman; “Copying all or parts of a program is as natural to a programmer as breathing, and as production. It ought to be free” (1984).

However, Stallman’s self-described “crusade” in the pursuit of these freedoms for all software developers and user should not be confused, he warns, with “non-commercial.” The ability to make a profit from one’s labor is a necessity: “Now, I wouldn’t have dedicated so many years of my life to making sure programmers got less money. That’s not my goal. I’m a programmer and I don’t mind getting money myself” (2001). But the location of economic profit in software itself is a mistake, according to Stallman; for software is “society’s most important resource.” Preventing people from using freely this resource by charging for its limited use and black-boxing the source code creates a “deliberately inflicted waste.” Because software is not a “physical object,” nothing is saved by making fewer copies of it, as in the case of automobiles, Stallman has noted. Instead what is wasted is the potential that comes from individual using, modifying, and redistributing the software. And it is this waste that software vendors generate by making their software proprietary; “for physical objects, of course, like cars, it is always going to take resources to make an additional one of them, each additional exemplar. But for software that’s not true. It takes no resources, except a tiny bit
of electricity. So there’s nothing we can save, no resource we’re going to allocate better by putting this financial disincentive on the use of the software.” Only by removing the economic value from the object of software itself can software be free. And only by creating free software is it made “possible for people to reject the chains that come with proprietary software” (1984).

Because proprietary software vendors deny both developers and users these morally rooted-democratic freedoms, or “inalienable rights” (2001), they act immorally and unethically. Stallman has continued to argue that having to “promise not to help my fellow hackers” through the use of proprietary software is not an option for those who believe in moral freedom. After all, software is not a commodity or property, and thus not intellectual property that must be protected, just as users of software are not consumers; “It is misleading,” Stallman has written, “to describe the users of free software, or the (sic) software users in general, as a ‘market’” (1999). In fact, the capitalist commodification of software is not appropriate to software because a product is not produced—only information.

As Stallman in an interview with *Byte* has explained, “The principle of capitalism is the idea that people manage to make money by producing things…But that doesn’t work when it comes to owning knowledge.” But Microsoft alone is not responsible for the stranglehold of proprietary software. Other proprietary software vendors are just as culpable: “I don’t care whether I’m chained by a big company like Microsoft, or a small company like Sun, or a start-up company or individuals, or whatever. I don’t want to have a master. I’m not willing to accept the chains, no matter who is holding them. We should focus on what Microsoft does to subjugate people, and not on the fact that they manage to do it to so many people” (as quoted in King). Thus, the construction of hackers and others who share free software as “pirates” is tantamount to “slander.” In fact, Stallman has urged individuals not to use “piracy” in conjunction with software; he writes, “If you don’t believe that illegal copying is just like kidnapping and murder, you might prefer not to use the word ‘piracy’ to describe it. Neutral terms such as ‘prohibited copying’ or ‘unauthorized copying’ are available for use instead. Some of us might even prefer to use a positive terms such as ‘sharing information with your neighbor’” (1996).
But it is not only vendors of proprietary software that Stallman positions himself against. Those, such as Torvalds, who look at free software as “a purely practical way of getting software written,” ignore the fact that “Free Software is a political action which places the principle of freedom above everything else” (2001). In doing so, pragmatists at best undervalue, at worst dismiss the importance of freedom to society as a whole. When asked about the use of proprietary software in the management of Linux’s source code, Stallman responded,

It is unfortunate that anyone uses proprietary software. Using it publicly for the development of a prominent free software package is particularly unfortunate, because it sets a bad example…There are already free programs that do the same basic job. Linus Torvalds feels they are not convenient enough…But just because we are competing with proprietary software on issues of technical merit doesn’t mean we think people should choose the program for source control based on technical qualities alone. That would mean assigning zero value to freedom itself. (as quoted in Biancuzzi, 2004)

But the politics of freedom and the need to recognize the value of freedom itself extend beyond the object of software. Torvalds’ use of “Linux” rather than “GNU/Linux” in referring to the OS that is in fact comprised of both Linux and GNU erases the political underpinnings of free software, according to Stallman who explained, “The ideas associated with Linux—the philosophy is very different. It is basically the apolitical philosophy of Linus Torvalds.” And for Stallman, free software is anything but apolitical. Consequently, he has advocated the use of “GNU/Linux” in order to draw attention to goals first outlined in his e-mail “new UNIX implementation”: “Our community’s strength rests on commitment to freedom and cooperation. Using the name GNU/Linux is a way for people to remind themselves and inform others of these goals” (2000). Because the pragmatic approach to software had yet to be articulated in such a way that Stallman could align himself and his audience with or against it, we can see that the pragmatic approach, though it also supports free software, is incongruent with the idealism put forth in his story “new UNIX implementation” and its subsequent reproduction and refinement, according to Stallman.
Software as a Tool: Torvalds’ Evangelism

Just as Gates and Stallman have continued to espouse trajectories for future action in keeping with the specific philosophies and freedoms constructed in their earliest archived narratives, so has Torvalds. The pragmatist trajectory for the software programming culture constructed by the narrative “Free minix-like kernel sources for 386-AT” in 1991 has since been reproduced and refined, most notably in what Torvalds has coined “Linus’s Law.” Linus’ Law initially invoked the scientific method as it applies to programming—“Given enough eyeballs, all bugs are shallow” (Raymond, 1999, p. 31) which translated means that with enough programmers using, testing, modifying, and redistributing modified source code all software bugs, or flaws, will be found, thus optimizing the quality of the code. Since then, Linus’ Law has continued to evolve and has emphasized software coding as both a social and pleasurable pursuit.

Torvalds, who from 1994 to 2004 worked as a software engineer for the chip and proprietary software vendor Transmeta in San Jose, CA, insists that three “motivational factors” drive people: 1. survival, 2. the social, and 3. entertainment (Torvalds, 2001; Torvalds & Diamond, 2001). According to Torvalds’ discussion of Linus’ Law, when survival, the most primal of motivations, is relatively assured, individuals are motivated to establish social relations with others—to be part of a community. To be entertained is the final motivation, and entertainment can take the form of playing chess, jumping out of a plane, or programming—whatever is of interest of interest to the individual in the pursuit of fun. Recounting his initial interest in the code that became the Linux kernel, Torvalds has explained his development of Linux was motivated in part by the need for entertainment: “I got into Linux because I was a technology geek…” (Torvalds & Diamond, p. 242). As my discussion of the nerd/geek factor in Chapter Three explained, programming is a source of fun for software-oriented technology geeks. Following his interest in programming subsequently brought socialization with other geeks as well as entertainment;

For me and for many other people, Linux has been a way to scratch two motivational itches at the same time. Taking survival for granted, Linux has brought people both the entertainment of an intellectual challenge and the social motivations associated with being part of creating it all. We may not have seen each other face-to-face very
much, but e-mail was much more than just a dry exchange of information. Bonds of friendship and other social ties can form over e-mail. (pp. 248-249)

As Torvalds has explained in his co-written autobiography *Just for Fun: The Story of an Accidental Revolutionary*, “I probably would have stopped [working on Linux] by the end of 1991…in a software kind of world I find that once you solve the fundamental problems of a project, it’s easy to lose interest. And that’s what was happening to me…Then two things happened to keep me going. First, I destroyed my Minix partition by mistake. Second, people kept sending me feedback” (p. 90). It was not profit or morality but personal interest, problem-solving, and community-building that fed the development of Linux, according to Torvalds.

But this is not to say that Linux was “just” a hobby. As Torvalds explains, “When I originally posted Linux, I felt that I was following in the footsteps of centuries of scientists and other academics who built their work on the foundations of others—on the shoulders of giants, in the words of Sir Isaac Newton” (pp. 93-94). Fellow programmers were constructed for Torvalds at the very beginning of Linux as contributors to and peer-reviewers of his work, as is the case in scientific communities. But rather than seeking to emulate the accomplishments of the greatest scientists, Torvalds has written, “I think of myself as an engineer, not as a visionary or ‘big thinker.’ I don’t have lofty goals. I just want to have fun making the best damn operating system I can” (as quoted in Diamond, 2003).

To this end, Torvalds, who owns the trademark for “Linux,” has continued, as Linux has become more popular, to emphasize the pragmatic signification of software as a tool. What matters most to Torvalds is making hardware work the way he wants it to through software. Consequently, Torvalds has resisted any assertion that his aim with Linux was to take down Microsoft.

…but Microsoft just isn’t relevant to what I do. That might sound strange, since they are clearly the dominant player in the market that Linux is in, but the thing is: I’m not in the “market.” I’m interested in Linux because of the technology, and Linux wasn’t started as any kind of rebellion against the “evil Microsoft empire.” Really I’m not out to destroy Microsoft. That will just be a completely unintentional side effect. (as quoted in Diamond, 2003)
As Torvalds wrote in his e-mail “new UNIX implementation,” what matters in the development and use of software is having the best tool to do the best job. Making software freely available, modifiable, and redistributable is for Torvalds the most likely way to do this. “Opening up one’s technology,” Torvalds has written, reveals “those benefits, all you have to do is just look at the comparatively low standards of quality of any closed software project” (p. 194). Consequently, Linux has been and will presumably continue to be about making the best quality software and increasingly that has meant embracing commercial interests; Torvalds has written, 

Without commercial interests, how else would Linux flow into new markets? How else would it create opportunities for innovations? How else would it be able to reach the people who want an alternative—a free alternative—to the bad technology that’s out there? What more realistic ways for open source to take hold than through the sponsorship of corporations? And what better way of getting some of the less interesting work accomplished, boring stuff like maintenance and support, than doing it inside companies? (Torvalds & Diamond, p. 164)

Though Torvalds, like Stallman, is not against commercial interest or ties to software, Torvalds has actually advocated creating close ties between commercial, even proprietary, software enterprises. Not surprisingly, the motivation has not been monetary profit but realism—Torvalds has spoken of turning down a $10 million offer from a “London entrepreneur” who wanted him to “lend my name to his fledgling Linux company as a board member” (p. 166). If the opportunity to work on or use Linux is to increase, and therefore provide both developers and users an increased opportunity to use the best tools in their computer needs, Linux has to be adopted by “new markets.” This pragmatic approach to increasing the dissemination of Linux emphasizes the importance of choice for developers and users in meeting their computer needs; the point of programs like Linux is, Torvalds has written, “about letting everybody play” (p. 164).

Torvalds has dismissed critiques by those who have a problem with free software partnering with corporations and failing to adopt an overtly political agenda, especially Stallman, whom Torvalds has called “the God of Free Software” (p. 58):
The thing that drives me crazy about Richard is that he sees everything in black and white. And that creates unnecessary political divisions. He never understands the viewpoint of anybody else. If he were into religion, you would call him a religious fanatic…

I admire Richard from afar for a bunch of reasons. And I guess I tend to respect people, like Richard, who have very strong moral opinions. But why can’t they keep these opinions to themselves? The think that I dislike the most is when people tell me what I should or should not do. I absolutely despise people who think they have any say over my personal decisions. (Torvalds & Diamond, pp. 195, 196)

Because Linux was not undertaken for “lofty reasons” (p. 194), Torvalds has continued to resist being co-opted either by those who reproduce entrepreneurial or idealistic approaches to software. To program according to rules imposed for no other reasons than they are rules only serves to limit the pragmatic potential of software; Torvalds has said, “I’m very much against unnecessary rules imposed by society” (p. 196). Even Stallman’s call for Linux to be referred to as “GNU/Linux” is something that Torvalds has found “irritating,” though not as much as Stallman’s insistence that all software should be free. As long as Torvalds is able to share his work with others and have them share theirs with him, he is satisfied. In this respect, Torvalds represents pragmatic software development in the same ways he had a decade before when his work was just beginning work on Linux. The construction of source code as a pragmatic project undertaken as play by programmers with a personal interest; shared in the spirit of cooperation and participation that is only possible through freedom; and developed, tested, and modified according to a scientific method has continued as his trajectory for future action for the software programming culture.

The reproduction of the trajectories of action and the sets of relations constructed through these trajectories in the respective narratives of Gates, Stallman, and Torvalds have remained relatively stable from their earliest moments of archived storytelling up to the present. Gates continues to advocate an entrepreneurial approach to programming that criminalizes the sharing, or piracy, of software and constructs software as a proprietary commodity, the economic value of which lies in the object itself. Resistant to this approach is Stallman who espouses the moral benefits of sharing modifiable, redistributable software,
while demonizing software vendors and their proprietary software as well as programmers who, though they may advocate the sharing of software, ignore the political implications of free software. Outside of the dichotomous set of relations (re)constructed by Gates and Stallman is the pragmatic trajectory of Torvalds who resists the dogmatic positions of both by emphasizing the need for the best methods of developing the best software tools for use in programming. For Torvalds, these methods include developing and sharing free software with not just a lack of demonization of proprietary software vendors but in many ways an embracing of them insofar as they benefit practically the dissemination and quality of Linux.

These competing, though not always antithetical, trajectories of future action have remained consistent in their efforts to stabilize and change the discursive coding of software. However, I would be remiss in suggesting that these trajectories of action and the sets of relations are attributable solely to Gates, Stallman, and Torvalds. As spokespersons they are effective only insofar as the actors for whom they speak authorize them to do so (Bourdieu, 1993). Consequently, to understand the ways in which actors consent to the representation of their desires, values, alliances, and struggles by spokespersons, I examine in the next chapter the role of identity in the network society and discuss the ways in which identity functions as a structuring resource for the building of the communities for which and through which Gates, Stallman, and Torvalds respectively speak. To this end, I seek to elaborate further the “multiplicity of force relations” (Foucault, 1979, p. 92) as they function in and through the software programming culture in relationship to larger issues of identity, and in doing so, to extend further the digital literacy of lay users in their recognition of the discursive coding of software in ideology.
CHAPTER FIVE: IMAGINING SOFTWARE THROUGH IDENTITY IN COMPETING COMMUNITIES OF THE SOFTWARE PROGRAMMING CULTURE

In this chapter, I again take up the question that I began to answer in the previous chapter: What are the political ideologies encoded in software? In the previous chapter, I analyzed the ways in which software and the cultural practices of the software programming culture are ideologically encoded through the narratives of the yet-to-be spokespersons Gates, Stallman, and Torvalds. I also argued that these texts construct, through each of these future spokesperson’s projective imagination (Emirbayer & Mische), trajectories of action for the software programming culture. These trajectories are each motivated by a particular desire on the part of the spokespersons and the communities for whom each came to speak. In this chapter, I now turn to the role of identity in shaping these ideological trajectories that have been reproduced and stabilized within the software programming culture. In doing so, I explain how it is possible for different communities to form within the software programming culture and for each community to imagine very different futures for themselves, for the software programming culture, and consequently, for lay users of software.

To this end, I first discuss the processes of identity building in the network society, as developed by Castells (1997). This discussion extends the significance of the social positionings of Gates, Stallman, and Torvalds as spokespersons within each community that grew, at least in part, around the always competing, though not always antithetical, discursive codings of software. By extending my discussion of the ideological trajectories to include the discourse of the communities within the software culture, I illustrate the collective evangelism of these trajectories. Such a discussion prevents assigning the discursive codings of software just to these spokespersons alone: Gates and his entrepreneurial narrative, Stallman and his idealistic narrative, or Torvalds and his pragmatic narrative. I raise the issue of shared identities and describe the ways these identities serve as “sources of meaning for actors” (Castells p. 7). I argue that the focus on evangelism illuminates the ways in which the projectivity of spokespersons and their communities are the result of actors bringing otherwise abstract identities and the structured sets of relations among those identities to
function in the everyday through the specific practices available to these actors in their roles as members of the software programming culture.

Rather than functioning as determinants of action, identities serve as resources from which actors within a social system, such as the software programming culture, can develop strategies for eventually “reconfiguring” (Emirbayer & Mische) the culturally-structured practices available to them and create the futures to which they aspire ideologically. (I discuss this reconfiguration through the deliberative action of “practical evaluation” in Chapter Six.) Shared identity within the software programming culture has resulted in the formation of communities dedicated to the creation of a future to which its members aspire. In the case of the software programming culture, I have identified these communities as the proprietary software industry, the Free Software community, and the Open Source community. But, the alignment that occurs within a community can also position a community with or against another community. Thus, the alignment of a community with or against other communities creates sets of relations. The creation of these sets of relations on a collective level further explains the “intervention of ideology into language” (Grossberg, 1996a, p. 137) that was revealed in the sets of relations constructed through the competing narratives of Gates, Stallman, and Torvalds.

But to understand the ways in which a community is formed through the coalescing of actors in their mutual support of a particular identity and its ideology demands knowing how a community participates in discursive coding. According to Keicolt (2000), “narrative is the most common source of identity work” (p. 121). Locating the narrative threads that tie a community together helps to reveal the social positioning of a community and its members against other communities. In keeping with the narrative constructions of their spokespersons, the proprietary software industry positions itself against the Free and Open Source communities. The Free Software community positions itself against the proprietary software industry but has a more complicated relationship with the Open Source community, which the Free Software community sometimes aligns itself with and sometimes aligns itself against. Likewise, the Open Source community positions itself to varying degrees of alignment with and against the proprietary software industry and the Free Software community. By understanding how these sets of relations are negotiated through projective
dimension of discourse manifest in these communities, lay users of technology can understand the complicated ways in which lay users are subject to these sets of relations through their software use. To this end, I now turn to Castells’ discussion of identity in the network society, which I have modified slightly as a result of my examination of trajectories espoused by the proprietary software industry, the Free Software community, and the Open Source community.

**IDENTITY IN THE NETWORK SOCIETY**

In his study of the synergy between technology and globalization in the formation of the network society, Castells (1997) disentangles the relations of power constructed through identity-building in the Information Age. Arguing that the conventional processes of identity formation in which an actor might claim multiple identities have eroded, Castells contends that an actor’s sense of self is increasingly situated around one primary identity. What may once have been considered the multiple identities of a single actor—mother, doctor, teacher, construction worker, Democrat—are instead “roles,” and these roles coalesce around a primary identity that provides a cohesive frame for the various roles claimed by an actor (1997, p. 7). Explaining the differences between identities and roles in the network society, Castells writes,

> Identities are sources of meaning for actors themselves, and by themselves, constructed through a process of individuation. Although...identities can also be originated from dominant institutions, they become identities only when and if social actors internalize them, and construct their meaning around this internalization.

> Identities are stronger sources of meaning than roles, because of the process of self-construction and individuation that they involve. In simple terms, identities organize the meaning while roles organize the functions. (p. 7)

This conceptualization emphasizes the ways in which identity provides an organizing frame around which actors in the network society construct meaning for themselves, their actions, and their world. Because meaning, as Castells defines it, is intricately tied to the assignment

---

36 Castells locates the reasons for this erosion of multiple identities of the self to the rise of global capitalism and “profit-maximizing” through commodification as well as to the dwindling of the nation state and “power-maximizing” through military and state apparatuses (1996, p. 16).
of “purpose” to an actor’s action, identity is also the source by which actors align themselves with other actors who share the same identity, though not necessarily the same role, and against those who do not.

Identity building in the network society has resulted, according to Castells, in three primary identities: the “legitimizing identity,” the “resistance identity,” and the “project identity” (pp. 8-10). The legitimizing identity is one of dominance in that it legitimizes the dominant institutions that structure society. As Castells explains, the legitimizing identity is “introduced by the dominant institutions of society to extend and rationalize their domination vis a vis social actors” (p. 8). As a result of the legitimizing identity, we have, in Gramscian terms, the “apparatuses” that make a “civil society” (Castells, p. 8). Though a civil society only exists through the people’s belief in it (i.e., in the belief in “for the people by the people”)—and by way of the legitimizing identities—the same society dominates people to continue what some might argue is its primary objective—its own reproduction. This “logic of domination” is a key feature to the legitimizing identity and those who internalize it.

Unlike actors who internalize the legitimizing identity and who thus espouse the benefits of domination in order to rationalize it, those with a “resistance identity” have ideological beliefs that run counter to the legitimizing identity. As a result, actors who internalize the resistance identity are typically marginalized, excluded, and/or stigmatized by society. Citing environmentalists, religious fundamentalists, feminists, gay-rights activists, and the Zapatista movement as examples of the resistance identity in particular roles, Castells argues that resistance identity-building is “generated by those actors that are in positions/conditions devalued and/or stigmatized by the logic of domination” (p. 9). As a result of dominant society’s stigmatization, actors with a resistance identity respond by “building trenches of resistance and survival on the basis of principles different from, or opposed to, those permeating society….” A collective formed through a shared “defensive” position aims to turn the tables on the collective with the legitimizing identity that excludes and marginalizes the collective with a resistance identity. The goal of the resistance identity is “the exclusion of the excluders by the excluded” (Castells p. 9).

Whereas the resistance identity seeks to turn the tables on legitimizing identity and the dominant structures of society that marginalizes it, the “project identity,” the third form
of identity that Castells discusses, seeks to get rid of the table completely. The project identity is identifiable “when social actors, on the basis of whichever cultural materials are available to them, build a new identity that redefines their position in society and, by so doing, seek the transformation of the overall social structure” (p. 8). Although stemming from a resistance identity, the project identity goes beyond defensive resistance by seeking to overturn the logic of domination that makes the legitimizing identity possible. Thus, the project identity aims to radically transform society so that the positions of domination and subordination no longer exist. Unlike the resistance identity, which aims to become the excluder, the project identity wants to transform the social structures that allow exclusion to occur, and, in the process, to change the way society exists. This aim differentiates the project identity from the resistance identity.

In the study at hand, we can then understand ‘software programmer’ as a role. But the ideological encoding both of software and of the iterative practices of the software programming culture that constitute that role result, to a great degree, not from the role itself but from each programmer’s identity, which exists independently of that role. It is identity that makes continuity possible in the various roles that an actor might have. However, it is only through a role of software programmer and the set of typified iterative practices that constitute that role that these identities can function in lived experience. Though identity and ideology are abstract, through the actions of actors both become rooted in reality. Identity ideologically shapes a role, and the role, with its specific iterative practices, constructs the identity. The effect of identity on roles is significant and helps to explain how actors who share both a specific identity and the role of software programmer can come to imagine and reify the same ideologically-laden trajectory for possible action. Likewise, in ideologically encoding software and the iterative practices of the software programming culture differently, the narratives of Gates, Stallman, and Torvalds for example, can be understood as reproducing in the software programming culture the multiplicity of forces that exist among the legitimizing identity, the resistance identity, and the project identity, respectively. However, the early narratives of these then yet-to-be spokespersons point to the ways in which alignment can exist. And these alignments point to positions that Castells’ identity taxonomy does not account for. As the subsequent examination of the ways in which the
competing discourse of the proprietary software industry, Free Software community, and the Open Source community shows, alignment can occur between a resistance identity and the legitimizing identity. To account for the sets of relations that are constructed through the ideological trajectories of software and the software programming culture, I speak of the identities that shape and are shaped by the discourse of the communities within the software programming culture as the “legitimizing identity;” “the resistant/would-be-project identity;” and the “resistant/would-be-legitimizing identity.”

IDENTITY IN THE SOFTWARE PROGRAMMING CULTURE

Identity is an issue that extends beyond any one actor, as Castells’ discussion illustrates. Thus, to locate the significance of the narratives of Gates, Stallman, and Torvalds, we must recognize that their significance, to a great extent, stems from the authorization and sanctioning of the discourse of each by the respective community for whom each is a spokesperson. To understand the network of actors and organizations that “stand behind” (Bourdieu, 1993, p. 249) and reproduce the meaning of software ideologically encoded by these spokespersons, I now illustrate the ways in which the meaning constructed in the entrepreneurial, idealistic, and pragmatic narratives are reproduced and reified through the storytelling of individuals and organizations. Together these individuals and organizations constitute different communities within the software programming culture: the proprietary software industry, the Free Software community, and the Open Source community. I argue that each of these communities is a result of the internalization of an identity that roughly corresponds to those discussed by Castells.

I identify the proprietary software industry as resulting from the coalescing of actors who have internalized the “legitimizing identity” because of this industry’s discursive reproduction of an entrepreneurial trajectory that reifies capitalism through established software technology and its marginalization of the Free and Open Source communities (see Figure 5.1) I identify the Free Software community as resulting from the coalescing of actors who have internalized a “resistant/would-be-project identity.” This community’s discursive reproduction of an idealistic trajectory seeks to resist both the capitalistic motive of the
Figure 5.1: Software Programming Communities
proprietary software industry by developing disruptive software technology encoded with the ideology of Kantian idealism and the Open Source community’s reification of the business world and its encoding of disruptive software technology according to the ideology of pragmatism. Finally, as Figure 5.1 illustrates, I identify the Open Source community as resulting from the coalescing of actors who have internalized “resistant/would-be-legitimizing identity.” This community’s discourse reproduces a pragmatic trajectory and resists the ways in which established software technology has been developed by the dominant proprietary software industry. The Open Source community does not condemn the social structure that values economic profit and it resists the Free Software community’s encoding of disruptive software technology and idealistic ideology.

Instead of locating the resistance identity as defensive only to the legitimizing identity, I locate the resistance identity as a liminal position. In this liminal position, the resistance identity can be resistant both to the legitimizing identity and the project identity. Locating the resistance identity liminally allows me to account for the spatial positions of these communities as always competing, though not always antithetical, to one another. By identifying the ways in which the proprietary software industry, the Free software community, and the Open Source community are aligned with and against one another, we can better understand the “multiplicity of force relations” (Foucault, 1979, p. 92) in the software programming culture. We can also interrogate the forces that extend beyond the software programming culture itself through the identities of the networked society of the Information Age.

**The Legitimating Identity and the Proprietary Software Industry**

The construction of software programming as the mass production of saleable commodities became dominant through the evangelism not only of Gates but also of many others who share the legitimizing identity. The entrepreneurial narrative and its encoding of what would eventually be established software technology reflected the economic profit-motive that came to dominate an ideological trajectory that served to rationalize the institution of capitalism for the software programming culture and lay users of software. This dominant ideology constructed a trajectory in which software for the PC “naturally” became a black-boxed commodity, programming became a paid profession rather than a hobby, and
monetary profit became the way in which to measure value. Consequently, the discursive coding of software by the proprietary software industry can be understood as resulting from the internalization of the legitimizing identity by actors and the organizations they created within the software programming culture.

Integral to the proprietary software industry are proprietary software vendors, who are primarily responsible for the development of the proprietary, black-boxed software that has become the established software technology for the software programming culture and society as a whole. Examples of proprietary software vendors include Microsoft, IBM, Sun Microsystems, Oracle, Adobe, and SCO. Not surprisingly, the vendor most successful in contributing to the encoding of software in the entrepreneurial narrative is Microsoft, whose very existence is an effect of the legitimizing identity internalized by co-founders Gates and Allen. From its formation in 1975, Microsoft has grown from a company with three employees and revenue of $160,000 to approximately 60,000 employees and revenue of almost $39.79 billion, the vast majority of which is generated from software sales (2005 Annual Report). As the world’s leading proprietary software vendor, Microsoft has continued to reproduce the very same ideological encoding of software found in Gates’ “An Open Letter to Hobbyists,” and, in doing so, has forwarded the entrepreneurial trajectory for software, the software programming culture, and its cultural practices.

Microsoft describes its trajectory for the future of software and the software programming culture in entrepreneurial terms: “We intend to sustain the long-term growth of our businesses through technological innovation, engineering excellence, and a commitment to delivering high-quality products and services to customers and partners [italics added]” (p. 21). But this trajectory is not constructed as the singular pursuit of economic profit from the sales of software commodities. Just as Gates constructed Allen, Davidoff, and himself as benefactors to hobbyists for the work they did on Altair BASIC, so too has Microsoft. In doing so, the proprietary software vendor not only rationalizes the dominant institution of capitalism but casts capitalism as a beneficial good, and, as a consequence, allows the legitimizing identity of the network society to function in the software programming culture. Indeed, Microsoft has constructed itself throughout the years as a benefactor by bringing much needed products to the personal computer market, by providing employment
opportunities, and, with its growing economic power and influence, by providing to society as a whole through the good works that Smith originally reasoned would be a natural effect of entrepreneurial “self-interest.” For example, in its 2005 Annual Report, Microsoft, in a rhetorical move that illustrates the breadth of the software market, describes its mission as follows:

> to enable people and businesses throughout the world to realize their full potential. We work to achieve our mission through technology that transforms the way people work, play, and communicate. Since our founding in 1975, we have been a leader in this transformation. We develop and market software, services and solutions that deliver new opportunity, convenience, and value to people’s lives. (p. 7)

Through the development of software as a proprietary commodity, Microsoft, according to this identity construction, has not just met the needs of a market but has essentially made most markets possible. With this possibility, Microsoft has also created the possibility for individuals and businesses alike to pursue their goals and, in doing so, to reap the value that arises from the attainment of those goals. Thus, the mission laid out in the entrepreneurial narrative reproduced by Microsoft and other proprietary software vendors is not a mission that benefits the software vendor alone. Instead, it is a mission that, in its success, carries with it the possibility of society as a whole through the beneficial work of corporations and their corporate citizens.

Another example of the legitimizing identity functioning through the entrepreneurial trajectory reproduced by the proprietary software vendor Microsoft is the cultivation of corporate partnerships in the capitalistic work of software programming. These partnerships function to strengthen and extend the logic of domination of the legitimizing identity. Just as Gates and Allen had formed a partnership with MITS, the manufacturer of the Altair 8800, Microsoft and other software vendors have formed corporate partnerships not only with other software vendors but also with original equipment manufacturers (OEMs). In 1975, Gates and Allen entered a partnership with MITS for the distribution of Altair BASIC. In 1980 Microsoft entered into what would be its name-making partnership with IBM for the software MS DOS and in 1982 with Apple for software compatible with Macintosh. Though other vendors have entered into similar partnerships, Microsoft was and continues to be the
industry leader. In fact, Microsoft’s partnerships with OEMs finally grew to such an extent that in April 2000 the company was found guilty of violating the Sherman Antitrust Act and was ordered to breakup into two companies, though the ruling was later thrown out. The alternate settlement, reached in October 2001 with the Justice Department, prevents Microsoft from punishing any hardware manufacturer for including non-Microsoft OSs. Though the OS GNU/Linux and other free and open software such as Firefox and Apache have certainly made inroads—Apache, in fact, leads the server market—many OEMs such as Dell, HP, and Gateway now include “recommendations” for Microsoft products on their corporate websites.

Corporate partnerships are not the only means by which the legitimizing identity and the entrepreneurial encoding of software is extended by the proprietary software industry. Microsoft acknowledges its indebtedness to the knowledge work of its employees in the success of its business: “Our business is based on successfully attracting and retaining talented employees” (p. 18). Just as Gates constructed software programming as a professional enterprise dedicated to the development of quality products from which Microsoft could cultivate revenue, invest in R&D, and provide better quality products to consumers through the work of professional programmers, so too does Microsoft 29 years later. This cyclic process allows Microsoft to continue to exercise its freedom to produce so that consumers can continue to exercise their freedom to consume: “Customers around the world continue to benefit from our innovations, while our development of new technologies will position the company for future growth” (p. 2). According to the trajectory constructed through the entrepreneurial narrative that allows the legitimizing identity to function in the software programming culture, future growth is dependent upon innovation, and innovation depends on the generation of revenue from current products. In 2005, Microsoft’s revenue allowed for an investment of $6.18 billion in R&D, or 15.5% of its revenue for the same year. This investment, made possible only through the construction of software as a commodifiable intellectual property brought Microsoft’s R&D budget for the last three years alone to $20.56 billion.

Because of the importance to the entrepreneurial trajectory of intellectual property, itself a construct of the dominant institutions of society and thus the legitimizing identity,
Microsoft has published a number of articles on its website that outline the advantages of intellectual property and the dangers of undermining it. Representative of these articles is “The Value of 1s and 0s,” an essay that appears in Microsoft’s collection on the 21st-century future of society in *Microsoft on the Issues: Essays on Society and Technology*. In the opening statements of this essay, Microsoft reaffirms the necessity of the entrepreneurial trajectory for the software programming culture for the next millennium, and in doing so, extends the logic of domination of the institution of capitalism to the 21st Century:

It is no coincidence that the United States—the world’s leading advocate for intellectual property rights—is also home to the world’s largest software industry. Strong intellectual property protections in the United States have also helped create thriving movie, publishing, recorded-, music and pharmaceutical industries which, combined with software, help drive the nation’s economy. Without these protections, creativity and innovation would be stifled, leading to fewer choices for consumers. Software development—like that of movies, music, books and pharmaceuticals—involves companies in any of the fields cannot be assured of a reasonable return on their investment, they are reluctant to develop new products. Many public officials, including Federal Reserve Chairman Alan Greenspan, have recently noted how information technology has helped improve America’s productivity. This, combined with falling prices, has helped cut the nation’s inflation rate, while overseas sales of American software products have provided one of the strongest recent bright spots in the United States trade balance. By next year, the software industry’s contribution to the United States economy will be greater than the contribution of any other manufacturing industry. (1999)

In discourse such as this, Microsoft intertwines its own interests in productivity and profitability with the national, and hence public, interests of the U.S. According to the entrepreneurial trajectory as a function of the legitimizing identity, only through the protection of intellectual property, whether software, digital content, or pharmaceuticals, are the interests of the nation and its economy served. Microsoft and other proprietary software vendors generate taxes, create employment opportunities, and support R&D. Through these
effects of capitalism in the proprietary software industry, the logic of domination is yet again constructed as rational and the legitimizing identity reified.

Yet, it is not only these overtly capitalistic enterprises that are advanced through the proprietary commodification of software. In a continuation of institution of capitalism as not just rational but beneficial to society, Microsoft is constructed as a humanitarian enterprise. As Smith suggested in *The Wealth of Nations*, the “self-interest of the entrepreneur promotes the public interest.” Though this promotion is certainly served by the overtly capitalistic enterprises of Microsoft and the proprietary software industry, the wealth generated by the production and consumption of software has made it possible for the rich of this industry “to contribute…relief” to the poor of society (p. 68). Most notably, the Bill & Melinda Gates Foundation has made grants to various charities and programs in the amount of almost $10 billion since its inception in 1994 (2004 Annual Report). With assets totaling $28.8 billion, the Gates foundation mission is described as follows by its president Patty Stonesifer:

Bill and Melinda Gates started their foundation because they want to increase equity—to help make sure that all people, no matter where they're born, have the chance to make the most of their lives. We've chosen four areas where we think we can do the most to promote equity: global health, U.S. education, access to digital information, and grantmaking in our local region. [http://www.gatesfoundation.org/nr/public/media/annualreports/annualreport04/letter/](http://www.gatesfoundation.org/nr/public/media/annualreports/annualreport04/letter/)

Funded by donations from Bill Gates’ personal fortune, which he amassed from the success of Microsoft and the acceptance of the entrepreneurial narrative by lay users of software technology, the Foundation supports the kind of philanthropy that is, in many ways, only possible because of successful entrepreneurialism.

However, in the reproduction of the construction of the proprietary software industry as a benefactor, the Gates Foundation is not the avenue by which this has been accomplished. As an effect of the legitimizing identity, Microsoft has constructed itself as a global citizen working towards the empowerment of others, a move that once again rationalizes the logic of domination supported by the legitimizing identity. In a joint executive letter, Gates and current CEO Steve Ballmer state, “At Microsoft, our passion to do well is matched by our
desire to do good. We believe the best way to achieve those parallel goals is to align our business and global citizenship strategies” (“Global Citizenship”).

In addition to its corporate donations to and support of NGOs, community-based organizations, and charitable government initiatives, Microsoft employees are characterized as an important part of the corporation’s global citizenship: “None of our achievements would be possible without the extraordinary efforts of our employees. Not only do they work hard every day to make Microsoft a successful business, but they also contribute their time, expertise, and personal resources to many of our citizenship efforts.” Included in these employee-related efforts are the organization of relief drives for earthquake victims in Taiwan, Columbia, and Turkey; the development of a system for registering Kosovo refugees; and donations to Microsoft’s Annual Giving Campaign, which, in 2004, totaled $12.5 million, which Microsoft then matched. Thus, with the expansion of the market for software that does not simply want quality software but is dependent upon it for the goings-on of everyday life, Microsoft—the corporation, its co-founder, and its employees—serves the public-interest by “stepping in to make a difference in people’s lives,” as Smith theorized would be possible in this “system of perfect liberty.”

Just as the entrepreneurial narrative of Gates and others in the proprietary software industry has emphasized the good that results in the professional programming of software as a proprietary commodity, so too has the entrepreneurial trajectory and its capitalistic ideology continued to equate non-proprietary software with lesser quality and the sharing of software with criminal activity. In doing so, proprietary software vendors, an effect of the legitimizing identity as it functions in the software programming culture, exclude through their discourse alternatives to the entrepreneurial narrative, alternatives that attempt to undermine the logic of domination. We have already seen that the same vision for software and the software programming culture is shared by Gates (the leading spokesperson for proprietary software and the industry that has sprung up around the entrepreneurial software-as-a-commodity approach), and Microsoft (the leading proprietary software vendor). But, with its corporate infrastructure, Microsoft need not rely solely on Gates or the faceless textual productions of Microsoft because a number of executives in the corporate hierarchy stand at the ready to
evangelize the benefits of proprietary software and the problems with any alternative to trajectory constructed by Gates in “An Open Letter to Hobbyists.”

CEO Steve Ballmer, who assumed the role in 2002, has been a vocal advocate of the entrepreneurial construction of software with its celebration of black-boxed proprietary software and the condemnation of software that runs counter to the entrepreneurial narrative. For example, in a speech to the Gartner Symposium and IText in Orlando, Florida, Ballmer spoke of the heightened quality and guaranteed value of the proprietary software when compared to shareable software:

I’m happy to tell you at all times why I think what we do is better than what any one of our competitors, commercial or noncommercial, open-source style competitors. And I think we do deliver more value, I think we can stand behind our products in a way that the open-source community can’t stand behind its products. (October 20, 2004)

In another challenge to the quality of open source software, Ballmer, in an interview with the technology-related online news site Cnet, challenged the innovativeness of Linux by equating it with UNIX: “[Linux] is a clone of an operating system that is 20-plus years old. That’s what it is. That is what you can get today, a clone of a 20-year old system. I’m not saying that it doesn’t have some place for some customers, but that is not an innovative proposition” (2003). With the continued reproduction of the entrepreneurial narrative in discourse such as Ballmer’s, proprietary software remains the quality work of professionals. After all, as Microsoft’s CEO reasons, Linux is a “weirdo competitor…because there’s no company behind it” (2003).

Not surprisingly, by advocating a “professional” approach by proprietary software vendors, evangelists who share a legitimizing identity and consequently espouse the benefits of proprietary software have also continued the negative characterization of those programmers and products outside the proprietary trajectory of software. In order to keep OEMs from supporting open-source software such as Linux, Joachim Keimpin, head of Microsoft’s OEM department, relayed a plan for hitting “the OEMs harder than in the past with anti-Linux” in a memo to Ballmer that was made public in the anti-trust case. Likewise, Microsoft vice-president Jim Allchin, in an interview with CNet, has argued, “Open source is
an intellectual-property destroyer” and as such is anti-American: “I’m an American. I believe in the American Way. I worry if the government encourages open source, and I don’t think we’ve done enough education of policy makers to understand the threat” (2001). Once again, we see the intertwining of the entrepreneurial trajectory for software and software programming culture with the national interests of the U.S., a process that contributes to the naturalization of the capitalist enterprise and software commodification.

In addition to the negative construction of non-proprietary software, Microsoft executives have also continued the association of “hacker,” a term that before the birth of the PC was linked with the practice of sharing software, with crime. In his testimony before the U.S. House of Representatives’ Subcommittee on Government Efficiency, Financial Management, and Intergovernmental Relations Committee on Government Reform, Scott Charney, Microsoft’s Chief Security Strategist, stated, “we are in a perpetual and accelerating race against hackers, and both the Government and industry need continuously to improve their cyber-security capabilities.” As Charney goes on to detail the way in which hackers have attacked both Linux and Microsoft software, (a move that reveals open source to be just as open to attacks as Microsoft in spite of evangelists claims that GNU/Linux and other free and open source software is less susceptible to viruses and worms), his use of “hacker” continues to conflate criminal behavior with the term that both free-software and open-source evangelists use to identify themselves.

This conflation is reproduced by watch-dog groups that protect the logic of domination of the legitimizing identity as it functions in the software programming culture. Such watch-dog groups include the Software Publishers Association, the Software & Information Industry Association, and the Business Software Alliance (BSA). The BSA, founded in 1988, is, in its own word, “the voice of the world's commercial software industry and its hardware partners before governments and in the international marketplace. Its members represent the fastest growing industry in the world.” The members of BSA include Adobe, Apple, Dell, HP, IBM, McAfee, Microsoft, SAP, and Symantec, to name just a few. In “promoting a safe and legal digitalized world,” BSA conducts studies, offers information on legal software use, and participates in the policing of illegal software sharing. In a speech in 2004, BSA President Robert Holleyman argued, “Determined, innovative hackers, virus writers, and cyber
criminals are constantly work to develop new ways of breaking into systems—just as criminals in the real world are continually inventing new types of fraud and finding new ways to break into cars or homes” (2004). In addition to the loss of information and privacy, the result of hacker criminals and other cybercrime perpetrators is the “chipping away at consumer confidence” in the proprietary software products users consume.

The BSA not only perpetuates the conflation of “hacker” with “criminal,” it equates the sharing of software with piracy, without acknowledging that the only kind of software that can be pirated is proprietary software. “Software theft is stealing,” the BSA explains in an informative report addressed to software programmers; “If you or your company would be caught copying software, you may be held liable under both civil and criminal law.” By conflating piracy, theft, stealing, and copying, the BSA continues the reproduction of the entrepreneurial narrative and illustrates the damaging effects of the theft of intellectual property. The BSA’s “Global Piracy Software Study,” undertaken with the research group IDC, reveals that though $90 billion in commercially packaged PC software was used in 2004, only $59 billion was actually spent on the purchase of that software (p. 1). The piracy of $31 billion worth of software denies revenue to software vendors, limits employment opportunities, potentially damages computers and data, and opens up users to criminal prosecution (“Anti-piracy Information, http://www.bsa.org/usa/antipiracy/). To combat the damaging effects of the illegal copying and use of software, the BSA operates approximately 65 international hotlines, including the U.S.’s 1-800-RU-LEGIT. These hotlines provide information about software piracy and allow individuals to report software piracy, as does the link “Report Piracy” on its website. By providing outlets to report the criminal activity of copying software, the BSA creates the opportunity for every person to police software users, whether an individual or a corporation, which illustrates the extension of the software market beyond the hobbyist club meeting that Gates argued needed policing in his “An Open Letter to Hobbyists.”

In examining the reproduction of Microsoft’s entrepreneurial narrative, itself an example of the software vendors that comprise, along with watch-dog organization such as the BSA, the proprietary software industry, we see why Gates has become an “authorized spokesperson” for this industry. Just as Gates has evangelized the benefits of the
entrepreneurial trajectory in the production and consumption of commodified software, so to have those organizations that both are an effect of and have an effect upon this trajectory, especially Microsoft, the Bill & Melinda Gates Foundation, and the BSA. Because of this verbal network constituted through the discourse of the proprietary software industry, we can understand how the legitimizing identity functions discursively in the software programming culture. This network serves to rationalize and extend the institution of capitalism through the rhetorical construction of commodified, black-boxed software and the proprietary software industry as not just rational but a benefit to programmers, lay users of software, the national economy, and those in need of charity. In espousing the benefits of encoding software according to the ideological trajectory of capitalistic entrepreneurialism, the proprietary software industry also constructs any alternative to this entrepreneurial construction as negative and, in some cases, criminal. It is against such an entrepreneurial trajectory and legitimizing identity espoused by Gates and the proprietary software industry that the resistance identities internalized by actors within the Free and Open Source communities struggle.

The Resistance Identity and the Free Software and Open Source Communities

I locate both the Free Software and Open Source communities as resulting from the coalescing of actors who have internalized a resistance identity. The resistance identity, according to Castells, is defensive, positioned against the legitimizing identity because of the marginalizing “principles” associated with non-dominant ideologies (Castells, p. 9). By identifying the ways in which the Free and Open Source communities are aligned and juxtaposed, we can better understand the “multiplicity of force relations” (Foucault, 1979, p. 92) as the function in the software programming culture through alignment and juxtaposition.

According to the projective trajectory constructed by Stallman, spokesperson for the Free Software community, we can understand how his idealistic encoding of software is contrary to the entrepreneurial trajectory of Gates and the subsequent reification of that trajectory through the discourse of the proprietary software industry. Stallman’s idealistic encoding is, in fact, a potentially revolutionary alternative to the narrative construction of software-as-commodity. By locating this idealistic projectivity within Castells’ discussion of identity, we can understand Stallman’s trajectory for software programming according to
morality rather than economic profit resulting from his internalization of a resistance identity. Because of the functioning of the legitimizing identity within this culture through the projectivity of Gates and the proprietary software industry, Stallman’s trajectory, in which software is shared freely according to the Golden Rule, is a function of the project identity, whose “principles [are] different from, or opposed to, those permeating society” (Castells, p. 9). Consequently, we can understand Stallman’s trajectory and the reproduction of the idealistic encoding of software as a rhetorical move not only to be resistant but also to overturn the capitalistic structure by which resistance is necessary. As such, I locate Stallman’s imagination of possible paths of future action for the software programming as framed by a resistant/would-be-project identity.

Torvalds’ pragmatic encoding of software can also be understood as resulting from his internalization of a resistance identity, in that Torvalds rejects the dogma of capitalistic entrepreneurialism and instead constructs software as a tool that should be shared so as to increase the likelihood of the success of that tool in solving practical programming problems. However, the defensive position resulting from this narrative construction in “Free minix-like kernel sources for 386-AT” is not a liminal position in which to become a project identity, as Stallman’s is. Instead, the pragmatic encoding of software is resistant to the manner in which entrepreneurialism is done. The capitalistic trajectory imagines software as a commodity from which to garner a monetary profit through its sale. Because of the narrative construction of this trajectory, the sharing of software becomes a criminal act. At the time in which Torvalds told his story of the development of the Linux kernel (the year 1991), this construction was and continues to be dominant. But, as Torvalds later discourse reveals, commercial interests are useful to the development of software as tool and are not inherently in conflict with the pragmatic encoding of software. As such, I locate Torvalds’ discourse as resulting from his internalization of a resistant/would-be-legitimizing identity. Torvalds is not opposed to Microsoft because it does business and reifies the institution of capitalism, but because of the way that it does business through black-boxed, proprietary software. Thus, we can understand the pragmatic encoding of software as a rhetorical move to displace the dominance of Microsoft, not because it is dominant but because its entrepreneurial encoding of software is values economic profit over the development of useful tools.
Consequently, we can understand the Free and Open Source communities, the former of which Stallman is spokesperson and the latter of which Torvalds is spokesperson, as having a shared defensive position against the proprietary software industry, a result of both spokespersons’ internalization of the resistance identity. However, complicating the alignment of these communities within a somewhat shared identity is the ideological incongruence of the idealistic and pragmatic narratives. This incongruence creates a disparity between the projective trajectories that the reproduction of the meaning constructed in these narratives cannot resolve. Drawing a distinct boundary between the trajectories constructed through the idealistic and pragmatic encodings of software, especially given that the latter grew out of the former, is sometimes not possible. At times, the idealistic trajectory is just as opposed to the pragmatic trajectory as the idealistic trajectory is to the entrepreneurial trajectory. Similarly, the pragmatic trajectory is just as opposed to the idealistic trajectory as the pragmatic trajectory is to the entrepreneurial trajectory.

To illustrate, grassroots activities in the evangelization of the disruptive software technologies of free and open source software include Linux user groups, which are geographically-located clubs (e.g., Potsdam, NY; Los Angeles, CA; Tel Aviv, Israel; Paris, France) that hold face-to-face meetings, and “Linux Install Fests.” These fests, which are intended to help Linux, and hence GNU, newbies install the GNU/Linux OS and other free and open-source software on their computers, do not allow for a neat categorization as an evangelist activity that is the result of either the idealism of Free Software Community or the pragmatism of the Open Source community. Because the ideological motivation behind those who seek to have Linux or GNU/Linux installed on their hardware and those who facilitate that installation cannot always be known, identifying these fests as an effect of either the idealistic or pragmatic encoding of software is not possible. The name “Linux Fests” certainly ignores Stallman’s and other free software evangelists’ call to use “GNU/Linux” rather than “Linux.” However, groups such as the GNU/Linux groups listed at the Free Software Foundation’s website that do heed Stallman’s call to refer to GNU—the software that with the Linux kernel constitutes an OS as GNU/Linux—could more conclusively be identified as part of the Free Software community, aligned with the idealistic trajectory, and internalizing a resistant/would-be-project identity.
In spite of the difficulty in identifying the distinction between the idealistic and pragmatic encodings because of the shared practice of freely sharing software, both the Free Software and Open Source communities can be identified as aligned through their shared opposition to the entrepreneurial narrative, the discourse through which the legitimizing identity has historically functioned in the software programming culture since the late 1970s. In my discussion of the discourse that reifies the respective ideological trajectory of each of these two communities, I locate their reconfiguration of the shared identity of resistance as resulting from two different moves towards a new identity: for the Free Software community this identity is the project identity; for the Open Source community, this identity is the legitimizing identity. Consequently, in a re-envisioning of Castells’ seemingly fixed notions of identity, I argue that the resistance identity is not itself necessarily an identity that is an end to itself but instead can be a liminal position between the legitimizing and project identities, as I believe is evidenced in the respective discourse of the Free Software and Open Source communities.

The Free Software Community as a Resistant/would-be-Project Identity

Illustrated in my analysis of Stallman’s narrative “new Unix implementation,” and its subsequent reproduction in his later discourse, the idealistic narrative sets for the software programming culture a moral trajectory that embraces the golden rule of the hacker ethic and rejects the primary pursuit of revenue generation through proprietary software production and consumption. The meaning constructed through this narrative and reproduced by the collective of the Free Software community can be understood as resistant to the logic of domination of the legitimizing identity perpetuated by the institution of capitalism as it functions through the proprietary software industry, the concept of intellectual property and the commodification of software. Accordingly, we can understand the Free Software community as a function of a resistant/would-be-project identity that, because of its wish to position humanity before profitability, seeks to transform the capitalistic structure of society and the logic of domination that flows through that structure.

For instance, the Free Software Foundation (FSF), established in 1985 by GNU Project founder and current FSF President Stallman, is a non-profit charity that “promotes the right to use, study, copy, modify and redistribute computer programs.” According to its
website, the FSF, which has European and Indian affiliates (FSF Europe and FSF India), “promotes the development and use of free software, particularly the GNU operating system, used widely in its GNU/Linux variant (http://www.fsf.org).” As the most prominent organization in the reproduction of the trajectory constructed through the idealistic narrative, the FSF, which is run by Stallman, an Executive Director, and a board of six directors, serves as one of the primary means of evangelism for the Free Software community and the functioning of the resistant/would-be-project identity in software programming culture. Supported primarily by tax-exempt donations, the FSF publishes and warehouses position papers and essays about free software; lists a directory of free software programs and tools; announces upcoming free-software events, including speeches by Stallman; and solicits support for free software campaigns, including software development and grassroots activities. In one such position paper, the FSF explains its philosophical trajectory and defensive position within the software programming culture by addressing the question “Is Microsoft the Great Satan?”

Many people think of Microsoft as the monster menace of the software industry. There is even a campaign to boycott Microsoft. This feeling has intensified since Microsoft expressed active hostility towards free software.

In the free software movement, our perspective is different. We see that Microsoft is doing something that is bad for software users: making software proprietary and thus denying users their rightful freedom.

But Microsoft is not alone in this; almost all software companies do the same thing to the users. If other companies manage to dominate fewer users than Microsoft, that is not for lack of trying.

This is not meant to excuse Microsoft. Rather, it is meant as a reminder that Microsoft is the natural development of a software industry based on dividing users and taking away their freedom. When criticizing Microsoft, we must not exonerate the other companies that also make proprietary software. At the FSF, we don't run any proprietary software---not from Microsoft or anyone else.

Once again, we have in the discourse of the FSF the rhetoric of democratic freedom. But, as illustrated in my analysis of the discourse of Free Software community’s spokesperson, this
construction of “freedom” is specific to the ability to act morally and to not deny any person the ability to do so. Because Microsoft and other proprietary software vendors are, in this idealistic construction, guilty of this denial, they also impede the idealistic trajectory reproduced by those within the software programming culture through their resistant/would-be-project identity.

Another function of the FSF that reproduces the idealistic encoding of software is the organization of grassroots campaigns that challenge and seek to overturn the entrepreneurial trajectory of the legitimizing identity as it functions in the software programming culture, much in the way that grassroots resistance movements in the 1960s did. For example, a web strike occurred from April 5th to April 15th, 2004. This strike, organized, in part, through the FSF website, was a protest against “attempts to legalise (sic) software patents in Europe.” Because patents are a common technique used by proprietary software vendors to gain, in the words of U.S. patent law, “the right to exclude others from making, using, offering for sale, or selling” an invention—in this case software—protesters, in effort to keep Europe from allowing software to be patented as the U.S. does, banned together, removed their own websites, and replaced them with an online demo banner for the duration of the protest (see Figure 5.2).

In addition to this online protest, a physical protest march was coordinated with the web strike and held on April 14th outside the European Council’s building in Brussels, Belgium, capital of the European Union (EU). The coordination of these protests, a common technique of a resistance by those in dominated positions, constructs the Free Software community as a movement that is not just defensive but potentially transformative in that campaigns such as the web strike aim to transform the dominant practices by which software is regulated, most notably patent law. The catalyst for this transformation can be identified as
resistance to “multinational” corporations and “patents lawyers” that function as apparatuses of the institution of capitalism in dominant society through their perpetuation of the concept of intellectual property and the EU “Council of Ministers,” which stands poised to join the multinationals and patent lawyers in denying users the freedom to act according to their moral wills by sharing software and its source code. Through such grassroots activities, the FSF works to overturn the logic of domination whereby patents are considered a necessary protection of proprietary software processes, specifically, and intellectual property, generally. As such, the FSF functions not only as a resistance identity in the software programming culture but also as a would-be-project identity in that the FSF’s defensive position strikes out at the legitimizing identity, as it functions through the proprietary software industry, in an effort to transform dominant society and its impediment of the idealistic trajectory.

But how can an idealistic community such as the Free Software community fund its activities? To fund the development of disruptive software technology that is idealistically encoded, the FSF accepts donations, just as Stallman has done with the GNU Project. These donations are solicited from like-minded individuals, who provide the majority of the
charity’s support in the form of associate membership dues to advance the moral mission of the Free Software Movement. To become an “associate” of the FSF, individuals are assessed $120 in dues. Although material benefits of associate membership are limited and, at times, rather kitschy (e.g., a hardcopy of Stallman’s collection of essays *Free Software Free Society*, an answering-machine greeting from Stallman, 20% discount off of FSF merchandise, an invitation to the Annual FSF Meeting held in Boston), the true benefit of associate membership is described as follows: “Most importantly, you can proudly proclaim that you are helping FSF carry out its mission to preserve, protect, and promote software freedom” (“Associate Benefits”). Of course, this freedom, as my analysis has illustrated, is a particular kind of freedom, a freedom rooted in Kantian idealism.

Beginning in March 2003, the FSF also began accepting donations from “corporate patrons.” Corporate patrons, who currently include Google, HP, IBM, MySQL, and Tivo to name a few, are assigned membership dues that are commensurate with the number of employees in the business; these dues range from $500 to $25,000. In return for their support, patrons are allowed to display the FSF patronage logo (see Figure 5.3) on their materials and website, receive consulting services from FSF, and are given two complimentary passes to FSF-sponsored events.

Figure 5.3: Free Software Foundation’s Patronage Logo

Though corporate patronage may seem to be at odds with the idealism of the Free Software community, this Kantian-like trajectory, even in Stallman’s narrative, was not anti-corporation per se—after all programmers need to make a living as Stallman explains in his “GNU Manifesto.” It is rather “anti-” the commodification of software and its source code by those corporations who economically subsist on the practice, and in doing so, profit economically at the expense of humanity. But to ensure the clarity of the relationship
between FSF and its patrons, the FSF website states, “Corporate patrons affiliate themselves with the FSF through financial support. The FSF does not endorse the activities of its corporate patrons” (“Patron Donations”). Consequently, the FSF does not necessarily support the endeavors of its patrons, and as such, remains committed to the democratic practice of sharing software for moral rather than business reasons. In 2003, the FSF had revenue of $750,000, with 78% of that revenue coming from associate members and corporate patrons.

The FSF is not the only organization through which a resistant/would-be-project identity functions in the software programming culture through the reproduction and reification of the idealistic trajectory. A number of free software distribution and service companies have been formed over the years. The most well-known of these companies is Cygnus Solutions, the first free-software distribution and service company. Founded by Michael Teimann in November 1989, Cygnus—notice the “gnu” in “Cygnus”—generates revenue not from the sale of proprietary software, but rather, through the sale of services related to free software such as software set-up, customization, and development through the “technical expertise” of its employees. In 2000, Cygnus was acquired by leading open-source distribution and service company Red Hat. The acquisition of Cygnus by Red Hat points to the difficulty in neatly dividing certain organizations affiliated with the Free Software community from the Open Source community, as I previously mentioned in my discussion of Linux Install Fests. And since its acquirement by Red Hat, Cygnus’ mission statement, published on its website, certainly seems to align itself with the trajectory of pragmatism rather than idealism: “Cygnus Solutions was created to provide businesses with the best possible computer product and services for the best possible value. We believe that providing a high quality product and maintaining a very thorough, thoughtful and productive staff will perpetuate growth and prosperity for our customers and us.” So why do I include Cygnus in a discussion of the Free Software community? I do so in order to point out that free-software companies began before “open source” was even formed in 1997 and to point to the Open Source community’s roots in the Free Software community, if not by ideology, then through the practice of sharing software.

To provide some background, in July 1997, a group of programmers associated with the Free Software community broke away from the movement. Concerned “that the Free
Software Foundation’s anti-business message was keeping the world at large from really appreciating free software” (DiBona et al., 1999, p. 3), these programmers adopted the term “open” to describe their pragmatic approach to software that respects the role of proprietary software while evangelizing the technical and economic benefits of open source. Since then, the Free Software community and its evangelists, including Stallman and the FSF, have been walking a fine line between differentiating themselves from the ideology of the Open Source community and identifying themselves with the success of the GNU/Linux OS. Most notably, these evangelists have done so by advocating what they consider to be the more inclusive and accurate name “GNU/Linux” instead of “Linux,” as it is the Linux kernel plus GNU Project software that constitutes the OS. By calling the OS “Linux,” free software evangelists argue the Open Source community and others that adopt this naming practice erase the place—both practical and ideological—of the GNU Project and other free software.

As stated on the FSF website,

We [of the Free Software Community] are not against the Open Source movement, but we don’t want to be lumped in with them. We acknowledge that they have contributed to our community, but we created this community, and we want people to know this. We want people to associate our achievements with our values and our philosophy, not with theirs. We want to be heard, not obscured behind a group with different views. (“Why ‘Free Software’ Is Better than ‘Open Source’”)

Consequently, we can understand the FSF and the like-minded idealist hackers who create and use free software and/or donate to the moral crusade described in the idealistic narrative as resistant not only to the legitimizing identity of the proprietary software industry but also to the pragmatism of the Open Source community, with which the Free Software community shares a defensive identity. Though Cygnus may have been founded before the advent of the Open Source community, its trajectory is clearly ideologically pragmatic. This teasing out of the differences that can and do occur in a shared resistance identity reveals social positioning of the Free Software community against the Open Source community. Though both communities evangelize the benefits of freely sharing software and its source code, the Free Software community does so in order to overturn the dominance of capitalism and its construction of economic profit as that which is valued most. As such, we can
understand this community as a function of a resistant/would-be-project identity within the software programming culture. But as the GNU/Linux naming issue and Cygnus examples illustrate, the Free Software community is also in a defensive position against the Open Source community, its pragmatic encoding of software, and its function as a resistant/would-be-legitimizing identity, as the following discussion of the this community illustrates in greater detail.

The Open Source Community as a Resistant/Would-Be-Legitimizing Identity

The pragmatic narrative and the Open Source community that evangelizes it function in the software programming culture as a result of the internalization of actors of a resistant/would-be-legitimizing identity. By understanding the community that authorizes and reproduces Torvalds’ discourse, we can understand how the pragmatic encoding of software is not the result of a lone actor’s narrative construction of a pragmatic trajectory for the software programming culture. It is, rather, the result of a community comprised of actors who share the same structured identity that exists beyond the software programming culture itself but can only function in the everyday through the discourse of actors like those in the Open Source community.

Like the Free Software community, members of this community locate themselves in a defensive position against the legitimizing identity as it functions through the proprietary software industry in the software programming culture. However, unlike the Free Software community, the Open Source community is not ideologically positioned against the legitimizing identity itself but rather against the way in which it currently functions through the discourse of commodified, black-boxed software. The Open Source community is open to software as a commercial enterprise, though one that sells customer service to support open-source software (like Cygnus) rather than software as proprietary intellectual property (like Microsoft). The Open Source community simply wants to change the ways business is done. Transforming an ideology that values monetary profit from the sale of commodities above all else into an ideology that values the sharing of knowledge in order to create better tools for better use, does not stand to overturn the institution of capitalism from its place of dominance—as the Free Software community wishes to do through its idealistic encoding of software. Consequently, the Open Source Community’s discourse is resistant to the
reactivation of the iterative practices of the culture by the proprietary software industry in its
development and evangelism of established software technology, but resistant to the ideology
of the Free Software community that encodes disruptive software technology ideologically.

Like the Free Software community’s FSF, the Open Source community has a
prominent advocacy organization—the Open Source Initiative (OSI). Run by an executive
board headed by President Michael Tiemann, this “non-profit corporation,” which accepts
donations, was established in February 1998 by former open-source evangelist Bruce Perens
and current OSI Emeriti board member Eric Raymond37 as a means of advertising to the
“commercial world” the pragmatism of open source. As explained on the OSI homepage:

The basic idea behind open source is very simple: When programmers can read,
redistribute, and modify the source code for a piece of software, the software evolves.
People improve it, people adapt it, people fix bugs. And this can happen at a speed
that, if one is used to the slow pace of conventional software development, seems
astonishing. (“Homepage”)

Because the world of software was and continues to be primarily proprietary, the OSI, in
order to succeed as proprietary software vendors have, must make the case that open source
is good for business too: “Open source software is an idea whose time has finally come. For
twenty years it has been building momentum in the technical cultures that built the Internet
and the World Wide Web. Now it's breaking out into the commercial world, and that's
changing all the rules.” Unlike the Free Software community, which emphasizes the benefit
of free software to humanity, the OSI tends to emphasize the benefit of open source to
commercial enterprises, especially outside of the software programming culture. We can
understand the identity that functions through the discourse of the Open Source community

---

37 Both Perens and Raymond are examples of the often-blurred but nonetheless ideologically distinct boundary between the
Free Software and Open Source communities. After joining the Open Source community, for which he wrote the meta-
license the Open Source Definition which I discuss at length in the following chapter, Perens left the OSI after becoming
uncomfortable with the overt commercialism of open source. In a telling exchange in a June 1999 interview in Wired
Magazine about the infighting within the Open Source community and Raymond’s decision to license the Open Source
trademark to technical book publisher O'Reilly & Associates for its Open Source Summit, at which non-open-source
products would be displayed, Perens answered had this to say about the incident and the identity of open source: “There's a
middle ground between free software and rampant Microsoft-like commercialism. The Open Source Initiative used to
occupy that ground. If you listen to Eric now, he's changed his tune: I just want to have software that doesn't suck. Eric has
been seduced.” To this comment, Raymond responded, “It's important to keep an eye on the goal, better software, rather
than on the particular means, open source. Most of us are cool with IBM and Sun being in the game. The anticommercial
minority is just marginalizing itself” (http://www.wired.com/wired/archive/7.06/).
as aligned with the capitalistic ideology of the legitimizing identity and against the idealistic ideology of resistant/would-be-project identity.

This can be most clearly seen in the OSI’s lack of a position on the issue of software patents. Whereas the FSF and the Free Software Community protest the use of patents because patents limit users’ morally democratic freedom to share, the OSI not only supports the right to have patents, but dismisses the Free Software Community’s morally-motivated rejection of patents: “The Open Source Initiative does not have a position on whether ideas can be owned, whether patents are good or bad, or any of the related controversies. We think the economic self-interest arguments for open source are strong enough that nobody needs to go on any moral crusades about it” (Homepage). Through such discourse, the OSI takes no position against regulatory practice that the proprietary software industry supports, even though this practice precludes sharing. At the same time, the OSI rejects the morally-driven, defensive rhetoric against patents of the Free Software community. In its continued effort to distinguish its role in the evangelism of transparent, non-proprietary software from that of the Free Software community, the OSI juxtaposes ideologically its position with that of the Free Software community in the following statement on its February 2006 FAQ page: “The Open Source Initiative is a marketing program for free software. It’s a pitch for ‘free software’ on solid pragmatic grounds rather than ideological tub-thumping. The winning substance has not changed, the losing attitude and symbolism have.” In a move to construct the pragmatic encoding of software as having no ideology at all, the OSI constructs the Free Software community as the only community motivated by ideology. The OSI, according to its discourse, is above ideology and cares only about the commercial adoption and use of the best software systems, applications, and tools. In constructing its position thus, the OSI, as an organization within the Open Source community, aligns itself with the ideology forwarded by the proprietary software industry as a function of the legitimizing identity: capitalism is good. And this is an ideological position that the Free Software community does not share because what is primary is capitalism, not humanity.

However, the OSI’s resistance to dogma is not limited to the Free Software community. The OSI has traditionally been a major outlet for stances against the entrepreneurial narrative articulated by the proprietary software industry, though the severity
of those stances has lessened as open-source software has gained corporate acceptance. For instance, on October 31, 1998, the OSI posted an internal Microsoft document that had been leaked to Raymond, “historian and resident ethnographer” of hacker culture (2001), contributor to the GNU Project, and prominent free and then open-source software evangelist whose essay “The Cathedral and the Bazaar” was a major factor in Netscape’s decision to make its web browser, Navigator, open.

As a prominent spokesperson in his own right, Raymond released the leaked document with his annotations to the national press, and the OSI posted what came to be known as the “Halloween Document” on its site. This Halloween Document, the first in a series of six leaked internal Microsoft memos and additional texts ranging in date from October 1998 to June 2004. These memos show that Microsoft did indeed consider the GNU/Linux OS, which the memo refers to as “Linux,” as a real competitor to Microsoft products. In this leaked memo, Microsoft describes “FUD [fear, uncertainty, doubt] tactics” that Microsoft might use to undermine Linux and other “open source” software. Additionally, the memo points to open-source tactics that Microsoft might borrow because of open source’s ability “to collect and harness the collective IQ of thousands of individuals across the Internet.” Originally, Raymond placed what he described as an “anti-Microsoft jeremiad” introduction to the first leaked internal memo about open source, what came to be known as “Halloween Document I.” In this “jeremiad,” which I quote at length below to demonstrate the ideological passion that does sometimes accompany the pragmatic encoding of software (in spite of the non-ideological construction of open-source software), Raymond writes,

These memoranda lay naked the assumptions of Microsoft’s corporate culture…Bill Gates has pissed me off from day one. I don’t mind that he got rich, but I do mind that he peddles himself as the ultimate hacker and God’s own gift to technology when his track record suggests that he wouldn’t know a decent design idea or a well-written hunk of code if it bit him in the face. He’s made his billions selling elaborately sugar-coated crap that runs like a pig on Quaaludes, crashes at the drop of an electron, and has set the competing world back by at least a decade….

Bill Gates pretends to defend “innovation,” and if he did I’d love him for it. But there’s very little evidence that Microsoft even knows what that word means…And
the absolute worst—the absolute worst—is that he’s conditioned computer users to expect and even love derivative, shoddily-implemented crap. Millions of people think that it’s right, it’s normal to have an operating system so fragile that it crashes three or four times a week and has to be rebooted every time you change anything deeper than the wallpaper. Dammit, we knew how to do better than that in 1975.

If you’re not an engineer, maybe you can’t understand how deeply offensive most techies find this kind of thing. It gives me actual pain to see what Microsoft has done to the computer world, to the expectations of users, to the craft of programming that I love…

Linus Torvalds jokes about world domination [via Linux], but Bill Gates means it. What Microsoft has done in the past is very bad, but I would not have gone to war with them over the past. The real issue is that they won’t leave me and my friends any safe place. They want to hijack the Internet we built with brains and sweat and blood; they want top-to-bottom control of computers everywhere; they’re determined to have it all.…

What is interesting about Raymond’s discourse, besides its affirmation of the defensive position that the Open Source community takes against the proprietary software industry, is that such a jeremiad was originally sanctioned by the OSI by allowing both Raymond’s introduction and the Halloween Documents to be placed on the site. Given that the OSI constructs the description of its mission almost exclusively in commercial terms, Raymond’s response clearly problematizes this construction. Moreover, with the subsequent adoption of open-source software as a viable option in the commercial world, thanks to a great degree to the discourse of the Open Source community that constructed open-source software as such, jeremiads such as Raymond’s became few and far between. In fact, by July 2000, Raymond’s introduction to Halloween Document I had been moved to his personal site, with an addendum stating that the OSI had asked him to do so. And in October 2005, the entire Halloween Document series was made to follow suit. Through this action, we can see the Open-Source community’s discursive move to distance itself from the idealistic ideology espoused by the Free software Community, which is a function of a resistant/would-be-project identity. At the same time, we can see the Open Source community’s move to align
itself with the entrepreneurial trajectory that is in line with the values of the legitimizing identity. Where the Open Source community does distance itself is with certain practices of the proprietary software industry, namely those reactivations that create what I have located as the established technology of software (e.g., black-boxed source code, linear development, etc.). Consequently, the Open Source community rejects the proprietary software industry’s way of doing business, but not business itself.

However, the OSI is not the only organization that reproduces and reifies the pragmatic encoding of software as a function of the Open Source community’s resistant/would-be-legitimizing identity. Founded in 2000, the Open Source Development Lab (OSDL), where Torvalds has worked since 2003 after leaving the proprietary software vendor Transmeta, is, according to its website, “a non-profit resource lab for open source developers that provides equipment and infrastructure to large-scale Linux enabling technology projects to support enterprise and telecom applications.” With locations in Portland, Oregon and Yokohama, Japan, the OSDL seeks to increase the use of Linux in large-scale business computing, commonly referred to as enterprise computing. Supported by donations from a global consortium of IT leaders that include Google, Hitachi, HP, IBM, Intel, Motorola, and Nokia, the OSDL, according to its website, states that its mission is “to accelerate the deployment of Linux for enterprise computing through:

- Enterprise-class testing and other technical support for the Linux development community.
- Marshalling of Linux-industry resources to focus investment on areas of greatest need thereby eliminating inhibitors to growth.
- Practical guidance to our members - vendors and end users alike - on working effectively with the Linux development community.”

As with the OSI, we can see in the OSDL’s mission the reproduction of the pragmatic trajectory constructed in Torvalds’ discourse. The OSDL emphasizes the development of tools and the providing of services. By making this development and support primary, the OSDL argues that partners and customers will inevitably receive best enterprise software for the computing needs of their enterprise computing needs. The support of donors ensures that this will be the case. By having the best software possible, open-source distributors, partners, and customers will be able to maximize their economic profits. Because open-source
distribution and service support are the ways in which open-source software directly contributes to capitalism, though certainly not in the substantial ways that the proprietary software industry currently does, open-source distribution and service vendors are integral to the ability of the Open Source community to align itself with legitimizing identity. One such vendor, who is also corporate member of the OSDL, is Red Hat, the most successful of the Linux vendors. Founded in 1994, Red Hat employed 940 employees as of February 2005 (Annual Report, p. 13) and generated revenue in the amount of $196.5 million in that same year (p. i), primarily through the distribution and services related to “Red Hat Enterprise Linux” (RHEL) subscriptions. For a fee, subscribers are provided with maintenance, including configuration services and software updates. These services and updates are the result of not only the work of the OSDL but also Red Hat’s investment in research and development, which totaled $32.4 million in 2005. Like the OSI, Red Hat constructs open source as not only “a viable alternative to traditional proprietary software” but also more useful to software developers (p. 4). The use-value of Linux and other open-source software

- “allows a company’s in-house development team to collaborate with a global community of independent developers;
- provides the user access to both binary and source code, and the rights to copy, modify, alter and redistribute the software; and
- permits the user ongoing access to improvements made to the software by others.”

In evangelizing the benefits of the pragmatic encoding of software, Red Hat has become the most successful Linux distribution and service vendor. Through Red Hat’s “strategic relationships,” built on the shared development and use of Linux and other open-source software by Dell, HP, IBM, Intel, and Oracle, Red Hat has succeeded in promoting the market expansion of RHEL (p. 7).

Nevertheless, these partnerships can also be the greatest source of market competition for open-source distribution and service vendors, though such competition is, as Adam Smith described, crucial to the functioning of capitalism and through this institution the function of the legitimizing identity within and outside of the software programming culture. As Red Hat notes: “With our professional service offerings, we additionally face competition in the market for services related to the deployment of our enterprise technologies, including
Our competitors in the market include IBM as well as other technology infrastructure consulting companies” (p. 10). Because the Open Source community constructs software as a practical tool first and foremost, the fact that competitors within the open-source model are also partners in the development of software and the sharing of its code is not seen as inherently problematic. In fact, such partnerships work very well in rationalizing the logic of domination as it functions through the legitimizing identity in the software programming culture. There is little need to rationalize the logic of domination if everyone joins in partnership and reproduces the same entrepreneurial trajectory.

These partnerships are integral to the creation of the scientific community that is, according to the pragmatic encoding of software, the Open Source community. An additional benefit to users of open-source software is the choice of distribution and service vendors that is a result of these mutual software development partnerships. Switching from one proprietary software vendor to another can be extremely costly for a company because it has to purchase an entirely new software infrastructure. With the emphasis on economic profit from service rather than commodity software, open-source distribution and service vendors can provide customers using open-source software with the ability to switch service vendors without having to overhaul their software infrastructure. And because multiple distribution and service providers are working together with organizations like the OSDL to develop software for their own needs and their customers, companies using open-source are more likely to receive the best software for their situations rather than the software and services that they are already locked by a proprietary software vendor such as Microsoft. As Red Hat explains in its answer to the question, “Why Open Source,”

Freedom means choice. Choice means power. That's why we believe open source is inevitable. It returns control to the customer. You can see the code, change it, learn from it. Bugs are more quickly found and fixed. And when customers don't like how one vendor is serving them, they can choose another without overhauling their infrastructure. No more technology lock-in. No more monopolies. (“Why Open Source”).
We can understand Red Hat’s emphasis on consumer choice as an emphasis on the capability of an open-source software vendor to solve consumer’s practical problems and provide solutions for consumer’s needs without locking them into their service and products. The openness of transparent, non-proprietary software, developed, tested, and modified by a community of programmers that extends beyond the borders of a particular vendor, allows a consumer to make the switch, an act that becomes nearly impossible when a consumer is already locked into a proprietary software system whose developer is the only vendor capable of offering any services for that software. By advocating a change in the way that the software programming culture has traditionally thought about the business of software (without rejecting or challenging the concept of software as primarily a business enterprise), the discourse of Red Hat, the OSI, the OSDL, and Torvalds constructs the Open Source community. This community reifies the dominant institution of capitalism and thus the legitimizing identity. But, because of the legitimizing identity’s current construction through the discourse of the proprietary software industry, the Open Source community is also resistant to the legitimizing identity as it currently functions in the software programming culture. Practically, the Open Source community is aligned with the Free Software community in that they both value the practice of freely sharing software and source code. However, ideologically these two communities could not be more different. This difference is what leads both the Free and Open Source communities not only to resist the proprietary software industry but also each other.

Considering the ways in which organizations and the collectives of actors that belong to these organizations authorize the discourse of various spokespersons, we can understand the threading, reproduction, and reification of the narrative constructions projected by Gates in the entrepreneurial narrative, Stallman in the idealistic narrative, and Torvalds in the pragmatic narrative, as constituting different communities within the software programming culture. Each of these communities aligns itself with and against a particular ideological trajectory(s) that I suggest can be identified as the reflective reconfiguration of a shared identity that is not specific only to this culture: the legitimizing identity; the resistant/would-be-project identity; and the resistant/would-be-legitimizing identity. Instead, these identities align collectives of actors beyond the particular set of shared practices that constitute the
various cultures in which any one actor may participate. In doing so, these identities create the possibility for active transformation on the part of actors through the resources provided to them in their roles through the interplay between these otherwise abstract configurations of identity and the specific iterations that provide the means through which these identities function in and across cultures.
CHAPTER SIX: RECONFIGURING COPYRIGHT AND SOFTWARE LICENSING AGREEMENTS AS STRATEGIC DELIBERATIVE ACTION

In this chapter, I return to the question I began to answer in Chapter Three: What are the practices by which these ideologies function in the software programming culture? I return to this question to analyze the ways in which actors creatively reconfigure a shared iterative practice according to their identities and ideological beliefs. Using the temporal frame offered by Emirbayer and Mische’s discussion of “practical evaluation” as “the capacity of actors to make practical and normative judgments among alternative possible trajectories of action, in response to the emerging demands, dilemmas, and ambiguities of presently evolving situation” (p. 971), I examine the ways in which the iterative cultural practice of copyright (as manifest in the genre of software licensing agreements) has been strategically reconfigured according to the respective “projectivity” of actors within the competing, though not always antithetical, communities of the proprietary software industry, the Free Software community, and the Open Source community. I also examine the ways in which these different applications legitimize the sets of relations created through the discursive constructions of the ideological trajectories of these communities. In sum, I do a genre analysis of a software licensing agreement associated with each of these communities to show what each community achieves through a specific application of the genre of licensing agreements.

As I explained in Chapter Three, iteration, according to Emirbayer and Mische, is “the selective reactivation by actors of past patterns of thought and action, as routinely incorporated in practical activity, thereby giving stability and order to social universes and helping to sustain identities, interactions, and institutions over time” (p. 971). In keeping with the spatio-temporal framework of which iteration is part, I identified and described a network of practices that together constitute the software programming culture and the ways in which these practices are “reactivated” differently, a process that led to the development of established and disruptive software technologies (Christensen). One of the practices that I identified in this network is the institutionalized practice of copyright, a practice that regulates software use and software users. I continued my analysis of the discursive coding of software in Chapters Four and Five by examining the ways in which actors and the
communities of which they are a part imagine trajectories of possible action through their
projectivity—what Emirbayer and Mishe define as the “possible future trajectories of
action, in which received structures of thought and action may be creatively reconfigured in
relation to actors’ hopes, fears, and desires for the future” (p. 971). In doing so, I identified
three always competing, though not always antithetical, ideological trajectories for the
software programming culture: the entrepreneurial, the ideal, and the pragmatic. I discussed
the respective reproduction of each trajectory by a community within the software
programming as a result of collective of actors within the software programming culture that
share an identity (Castells) that extends beyond the culture itself: the proprietary software
industry as the legitimizing identity; the Free Software community as a resistant/would-be-
project identity; and the Open Source community as a resistant/would-be-legitimizing
identity. By identifying the shared cultural practices and examples of their reactivations as
well as the ideological trajectories evangelized by the different communities and their
spokespersons, I created a context through which to understand different applications of the
genre of software licensing agreements as the result of deliberate action.

To this end, I undertake in this chapter a close analysis of the way in which an
iterative practice is reconfigured according to the projective imagination of actors in the
various communities of the software programming culture. Specifically, I conduct a genre
analysis of the typified applications of copyright in the software programming culture and
argue that the differences in these applications are a result of deliberate rhetorical action
mediated by the temporal orientations of iteration and projectivity as well as the spatial
positionings organized around both. By analyzing these software licensing agreements
according to genre, I am analyzing, in the words of Miller, “typified rhetorical actions based
in recurrent situations” (1984, p. 159). The recurring situation that provides an exigency for a
software licensing agreement is the first-time use of a specific software program (e.g., an
operating system or application). The typified response to this situation is an agreement
between the new user and the copyright owner of the software. Understanding software
licensing agreements as a genre pushes me to examine how these agreements are “both
means and end, both resource and product” (Miller, 1994, p. 70). Conducting this analysis in
the context of iteration, projectivity, and the social positionings constructed through each
encourages the recognition of the ways in which the genre of software licensing agreements mediate “private intentions and social exigence” (Miller 1984, p. 163), a mediation that reflects practical evaluation as deliberate action. As Emirbayer and Mische explain, “by increasing their capacity for practical evaluation, actors strengthen their ability to exercise agency in a mediating fashion, enabling them (at least potentially) to pursue projects in ways that may challenge and transform the situational contexts of action themselves…” (p. 994). However, I also suggest that in this study of the discursive coding of software by the software programming culture, articulating the effects of this mediation can also benefit those who will never be members of this culture—the lay user of software. By understanding the ways in which deliberate action is mediated and reproduces sets of relations that are a result of ideological disparity, I argue that lay users of software can understand how software is always already discursively coded. In doing so, we increase our digital literacy so that our own software use is the result of reflective judgment and deliberative action. After all, whenever lay users click “accept” to a software licensing agreement for a piece of software they wish to use, these users are directly and legally subject to the discursive coding of software and the sets of relations created through that coding.

THE GENERIC APPLICATION OF SOFTWARE LICENSING AGREEMENTS

Licensing agreements sanction and regulate a licensee’s, and hence, user’s, legal ability to copy, distribute, and modify software as well as to receive a warranty for the licensed software. These features as the basic generic structure of software licensing agreements. I suggest that by analyzing the application of this structure in particular licensing agreements associated with the different communities of the software programming culture, and thus the competing ideological trajectories of each, we can understand differences and similarities as the result of deliberate rhetorical action on the part of the licensors and the communities that each evangelizes in a particular license with which they are ideologically aligned. In this section, I conduct an analysis of the regulatory practice of software licensing agreements and the ways in which this practice is mediated within the software programming culture by the structured iterative practice of copyright and the projective ideological trajectories of the communities whose software use is regulated by these agreements.
Each of the software licensing agreements that I analyze—Microsoft’s End-User Licensing Agreement (EULA) and the Free Software Foundation’s GNU General Public License (GPL)—grants licensees the right to use, not own, the software, provided they agree to the rights and limitations specified. Because a software licensing agreement is a contract between the licensor (the owner of the copyright for a program) and the licensee (an authorized user of a program whose copyright is owned by someone else), the licensee must agree to the rights and limitations specified in the agreement in order to use the software legally. Otherwise, the licensee is not granted the right to use the software at all.

Through an analysis of these licensing agreements, I suggest we can understand how these agreements set and regulate the conditions for software copying, distribution, modification and warranty in ways that correspond to the always competing, though not always antithetical, ideological trajectories of the proprietary software industry and the Free Software communities. In fact, these different applications of software licensing agreements cement in law the discursive coding of software evangelized by these communities as well as the sets of relations between these communities. In addition to these licensing agreements, I also analyze the meta-license of the Open Source Initiative—the Open Source Definition (OSD)—to illustrate how the Open Source community seeks to set the conditions through which software licensing agreements make material the pragmatic trajectory in a rhetorical move that practically aligns this community with the Free Software community, while, at the same time, ideologically aligns the Open Source community more overtly with the proprietary software industry. Though the OSD, as a meta-license, is somewhat different in its purpose in that it regulates what may or may not be considered an open-source license, the OSD does have the same generic conventions as Microsoft’s EULA and the Free Software community’s GNU GPL, also referred to simply as the GPL. The conventions include copying, distribution, modification, and warranties. Consequently, I analyze the OSD as part of the genre of software licensing agreements.

Before beginning my analysis, let me first summarize my argument on the EULA, the GPL, and the OSD and map through Figure 6.1 (next page) the relationship of each of these licensing agreements to the relationships I have mapped thus far. The EULA basically
Figure 6.1: Software Licensing Agreements as Deliberative Action
says that in order to use the software for which Microsoft holds the copyright, the licensee must not redistribute that software to anyone else; to do so is stealing and punishable by law. Microsoft software is an “established software,” meaning, as Figure 6.1 illustrates, that the reactivation of the shared cultural practices results in black-boxed software (i.e., it does not provide source code to users). Consequently, the EULA, in its reification of the entrepreneurial narrative constructed by Gates and Microsoft, as well as reified by the proprietary software industry and its legitimizing identity, does not give a licensee the right to the blueprints for this proprietary software vendor’s intellectual property, from which Microsoft generates economic profit. But what is included as a right to an EULA licensee is the right to a “limited warranty,” a common consumer “protection” in capitalistic society. These rights and limitations constructed through the EULA make possible the continued production of established software technology and reify the social positioning of Microsoft constructed through the discourse of Gates and Microsoft as a proprietary software vendor. Consequently, the EULA serves to extend Microsoft and the legitimizing identity’s dominance and to legitimate the position of Microsoft, as the copyright holder (licensor), against actors and organizations within the Free and Open Source communities.

In contrast to the EULA, the GPL, which the Free Software Foundation claims is “the most widely used Free Software license,” basically states that licensees are given the same rights to copy, modify, and distribute GPL-regulated software as the licensor, a feature commonly referred to as copyleft. Through copyleft, the freedom to share software and its source code is inscribed in law. This right is not only legal but, more importantly, moral according to the idealistic narrative constructed by Stallman and reified by the Free Software community and its resistant/would-be-project identity (Figure 6.1). And it is this right that allows disruptive software technology to be developed, used, and prevented from being appropriated by those who might try to make the GPL-regulated software proprietary. The GPL stipulates that source code must be provided to a licensee by the licensor. Failure to do so violates the GPL. In fact, attempting to deny other licensees of GPL-regulated software and any of its modified versions is a violation and results in the rescinding of the rights provided to the licensee by the GPL. According to the idealistic trajectory evangelized by the Free Software community, software developers and users should never been denied the rights to
share, modify, and redistribute software, nor should the accompaniment of the software with a warranty provided by the licensor be regulated by the GPL. Consequently, the GPL ensures this right by inverting the traditional use of copyright that typically denies licensees the rights that the licensor enjoys. Through the copyleft feature of the GPL, licensees are given the same rights as the licensor and prevented by law from doing otherwise.

The OSD, which author Bruce Perens (1999) describes as a “derivative” of the GPL (p. 172), basically says that in order for a software licensing agreement to be sanctioned by the Open Source community’s flagship organization, the Open Source Initiative, as regulating true open-source software, the agreement must include certain rights and limitations. These rights and limitations, which reify the pragmatic trajectory of Torvalds and the Open Source community as a resistant/would-be-legitimizing identity, are similar to those of the GPL. Licensees are given the right to copy, modify, and redistribute software. In order to take full advantage of these rights, the OSD states that a licensing agreement must insist that source code be provided with the software. However, these rights are not stipulated so as to enact an idealistic trajectory for the software programming culture and lay users of software, but rather to encourage the creation of quality disruptive software technology and to extend licensees’ right to choose, two objectives that are integral to the pragmatic trajectory. In keeping with the right to choose, the OSD does not stipulate whether or not an open-source software licensing agreement can include a warranty. The choice is up to the licensor.

By conducting an analysis of the generic applications of Microsoft’s EULA, the Open Source community’s OSD, and the Free Software Foundation’s OSD, I illustrate the ways in which proprietary, free, and open-source software are regulated by law according to both the structure of copyright as well as by the competing trajectories evangelized by various communities of the software programming culture. Each of the application of these agreements and meta-license, according to the conventions of the genre (e.g., copying, modification, distribution, and warranty), represents a strategic rhetorical move to legitimate a particular ideological trajectory and to have software use reify that trajectory.
Copying and Distribution

The copying and distribution features of software licensing agreements specify the rights and limitations concerning the ability to install software copies, to make additional copies of software for archival use or for use on additional computers, and to distribute software to others. These basic generic features, which are common to all software licensing agreements, grant a licensee the ability to copy and distribute legally licensed software in the ways specified by the licensing agreement. As my analysis illustrates, the copying and distribution features of the EULA differ greatly from those of the GPL and the OSD. However, even though the GPL and OSD share similar applications of these features, the language of these licensing agreements reify the differences between the Free Software community’s idealistic trajectory and the Open Source community’s pragmatic trajectory.

Microsoft’s EULA

The strategic rhetorical choices reflected in the Microsoft’s EULA are, I argue, a result of the deliberate discursive action to cement in law for the software programming culture, as well as lay users, the ideological trajectory of the legitimizing identity as it functions in the entrepreneurial narrative of the proprietary software industry. Proprietary software licensing agreements, such as the Microsoft EULA, cements in law the entrepreneurial construction of software as a commodity from which economic capital is to be accrued. Because the institution of capitalism, rationalized by the legitimizing identity of which Microsoft and the proprietary software industry are a function, traditionally locates the value of a consumable product in its monetary sale or retail value, the EULA limits a licensee to a prescribed number of non-distributable copies and use of those copies so as to maximize the amount of capital that can be accrued by the copyright owner from each copy of the software. Consequently, the EULA serves as a legal technique that protects both the exclusionary rights of Microsoft as the inventor and copyright-holder of proprietary software products and the value of the commodity being licensed, not sold, for limited use. Simultaneously, the EULA instantiates both within and outside of the software programming culture the conditions for illegal copying and distribution of commodified software by designating what rights users, whether legally licensed or not, are given or denied. In designating what rights are given to a licensee, the EULA also designates the denial of
certain rights for users who are or are not licensed to use the software. By doing so, Microsoft, through the EULA, aims to legally prevent others who do not enter into the contract of the licensing agreement from benefiting from the use of Microsoft commodities and from accruing capital from these commodities. Additionally, the EULA allows Microsoft, in its alignment with the dominant institutions that reify the legitimizing identity, to benefit from the punishments inscribed in U.S. and European copyright law, namely copyright infringement.

Specifically, the EULA, through the generic feature of copying, limits the licensee to a specific number of copies and use of those copies; this rhetorical move legitimates the entrepreneurial commodification of software and ensures that those constructions of software, professional software programming, the producer/consumer relationship imagined in Gates’ “An Open Letter to Hobbyists” and subsequent discourse both Gates as spokesperson and the proprietary software industry are legally binding. As explained in the EULA, “You [the licensee] may install, use, access, display and run one copy of the SOFTWARE on the COMPUTER.” With the exception of volume licensing—in which a business, government organization, or academic institution is given the right to install a copy of the software on multiple computers—the Microsoft EULA, the terms of which are common not only to Microsoft’s proprietary software but to most proprietary vendors, limits the copying and use of software independently purchased by an individual for personal use to one computer. As explained in the EULA, a licensee is allowed to make one back-up copy of the software in cases where the software comes pre-installed on a computer by an original equipment manufacturer (OEM):

YOU MAY MAKE A SINGLE BACK-UP COPY OF THE SOFTWARE. YOU MAY USE ONE (1) BACK-UP COPY SOLELY FOR YOUR ARCHIVAL PURPOSES AND TO REINSTALL THE SOFTWARE ON THE COMPUTER. EXCEPT AS EXPRESSLY PROVIDED IN THIS EULA OR BY LOCAL LAW, YOU MAY NOT OTHERWISE MAKE COPIES OF THE SOFTWARE, INCLUDING THE PRINTED MATERIALS ACCOMPANYING THE SOFTWARE.
Through this limitation, Microsoft, as the inventor and copyright-holder of the software licensed through the EULA, prevents licensees from copying onto more than one computer its copyrighted invention, unless otherwise specified in a volume licensing agreement. Doing so allows Microsoft to increase the market from the number of individual users to the number of computers, thereby maximizing the number of licenses that have to be purchased as an individual may own more than one computer. If a consumer does not agree to this or any other specification in the EULA, the consumer is advised by the EULA to return the “PRODUCT” for a “REFUND IN ACCORDANCE WITH MANUFACTURER'S RETURN POLICIES.”

Because the accrual of capital from commodified software, signified in the EULA’s discussion of a “REFUND” or return for payment, is dependent upon the number of licenses consumed, the legal binding of a license to a computer rather than to an individual licensee extends the market and hence the possibility for increased revenue generation. According to the entrepreneurial narrative of software evangelized by the Microsoft and other members of the proprietary software industry, this process of market extension benefits not just the entrepreneurially-driven producer but also those seeking employment as professional programmers, the nation to whom taxes paid, and all those who receive donations of time and money from the corporate citizenry. As explained in a discussion of its licensing agreements on its company website, Microsoft states, “To put it simply, [a license] is ‘married’ to the original PC on which it was installed” (Licensing Home), and thus, cannot not be copied onto a licensee’s other computers. To divorce a license from its legal spouse is consequently a denial of the democratic rights of the licensor, who, in the exercise of freedom, is legally entitled to demand and receive monetary payment for use of the software as intellectual property as well as all those who benefit from the institution of capitalism. Additionally, this denial of the right to copy a Microsoft software commodity at will creates the need for volume licensing agreements for businesses, academic institutions, and government and non-governmental organizations, which, without the entrepreneurial construction and legitimization of software, would otherwise be free to copy software onto all of their computers at will and without compensation to the licensor of the software.
Closely intertwined with the copying feature of software licensing agreements is the distribution feature. Because the entrepreneurial construction of software signifies software as intellectual property invented and owned by an entrepreneur or corporation, software enjoys the same protection as any other invention. Copyright law does not allow one person to copy or distribute unregulated an invention owned by another person because the value of that invention, in its copying and distribution, belongs to the inventor/copyright-holder and no one else. Consequently, the EULA, through its specification of the rights and limitations relating to distribution, protects the value of Microsoft’s intellectual property by denying others the right to distribute original copies or illegally-made copies of Microsoft commodities. To further the protection legislated in its copying feature, and thereby extend its right to have the commodity it produces consumed according to the number of computers in the market rather than the number of computer users, Microsoft, through the legally-binding contract of the EULA, denies users, whether licensed or not, the right to distribute its intellectual property, including the legally licensed software and the legally-made back-up copy. As explained in the EULA, “YOU MAY NOT LOAN, RENT, LEASE, LEND OR OTHERWISE TRANSFER THE CD OR BACK-UP COPY TO ANOTHER USER.” Consequently, Microsoft’s software commodities are not distributable under any circumstance. Because the right to make and use a back-up copy is limited to the licensee and the licensee only, the sharing of that copy with other users is not a right granted to the licensee, and is, in fact, illegal under the conditions instantiated by the EULA.

With proprietary software, the economic value of the software is located in the commodity of software itself, specifically in its sale value. Recall from the discussion in the previous, economic discussions of closed-source proprietary software emphasizes the sale value of the product as a “final good” (Raymond, 2001, p. 48). Consequently, the limitation on distribution, as specified in the EULA, maximizes the opportunities to capitalize on commodified software’s sale value, just as the copying feature does. A licensee who is granted the right to install a copy of purchased software on one computer or to make a back-up copy of software pre-installed by an OEM is limited as to what she can do with that software. Because distributing (i.e., sharing) a copy of Microsoft’s intellectual property with others denies, according to the entrepreneurial construction of software, Microsoft its right to
accrue capital from the sale value of its software, copyright law deems such actions illegal, unless otherwise stipulated in a licensing agreement. By using copyright law to protect its property by limiting both the ability to copy and distribute legally- or illegally-copied software, Microsoft, through the creative application of copyright, creates the possibility for users to act legally according to the rights and limitations regarding copying and distribution or to act illegally, and thus be guilty of copyright infringement. Ignoring or acting contrary to this limitation is under copyright law an infringement upon Microsoft’s exclusionary right to copy and distribute copies of its proprietary intellectual property.

To help users ensure that their software use is legally licensed, the EULA specifies that an identifying label known as the “Certificate of Authenticity” (COA) must be attached either to the packaging the software comes in or to the computer upon which the OEM installed the software:

If you acquired the SOFTWARE on a device, or on a compact disc or other media, a genuine Microsoft "Proof of License" COA label with a genuine copy of the SOFTWARE identifies a licensed copy of the SOFTWARE. To be valid, the label must be affixed to the COMPUTER, or appear on the SOFTWARE packaging. If you receive the label separately, it is invalid. You should keep the label on the COMPUTER or packaging to prove that you are licensed to use the SOFTWARE.

Through the EULA’s COA, Microsoft makes three assurances to a licensee. First, because proprietary software is produced and sold by a particular software vendor or its OEM partners, Microsoft as the vendor ensures with the proof of license that each copy of the software purchased or installed is an authentic copy. Second, because programming done by professionals ensures the quality of the software—an association first made in Gates “An Open Letter to Hobbyists and later espoused by many in the proprietary software industry (see Chapters Four and Five)—the COA also ensures that an authentic copy of commodified Microsoft software is a quality product. Third, by certifying the authenticity of the software, the vendor also ensures the legal use of the software, provided the licensee agrees to the licensing agreement, and the user’s knowledge of its legal status. Without a COA, the proprietary software is marked as inauthentic, and thus, is a warning sign to the individual that she does not have a valid license to use the software. Because unlicensed use of
proprietary software is constructed both through the entrepreneurial narrative and copyright law as illegal, a user is acting criminally.

According to this analysis of the generic features of copying and distribution in Microsoft’s EULA, we can understand this regulatory construction of software as deliberate rhetorical action mediated by the interplay of the structured iterative practices of the software programming culture and the ideological trajectory of the entrepreneurial narrative reproduced by both Gates and Microsoft. Copyright is regulatory practice explicitly defined by the U.S. government and sanctioned as applicable to software (see Chapter Three). And the application of copyright in the genre of software licensing agreements is a cultural practice shared by the competing communities of the software programming culture. The ideological trajectory that constructs software as a commodity and intellectual property is itself a reconfiguration by the proprietary software industry of the legitimizing identity that reifies the dominant institution of capitalism and constructs ownership, property, and capital accretment as the natural, logical rationale for society. Thus, we can understand the EULA as a technique that holds not only programmers but also lay users of technology subject to the entrepreneurial trajectory and its construction of democracy. This trajectory values a restricted relationship between consumer and producer so that the freedom to consume is secondary to the freedom to produce a commodity for economic profit, the maximum amount of capital that can be generated from the sale value of a software commodity. This maximization benefits the producer, the consumer, and society as whole through the legal copying and distribution of software licensed according to the rights and limitations specified in proprietary, black-boxed software licensing agreements such as Microsoft’s EULA.

**GNU GPL**

The GNU GPL is an application of copyright that reifies the values laid out in the idealistic narrative of the Free Software community. In the previous chapter, I located the Free Software community as a reconfiguration of the resistant/would-be-project identity. This identity, which positions itself defensively against the legitimizing identity, has two purposes: 1. to struggle against the logic of dominant society that embraces institutions that necessarily exclude and marginalize certain factions of society, and 2. to transform societal structures so that such exclusions no longer occur. Within the software programming culture,
this struggle for transformation manifests most notably in the Free Software community’s
defensive position against the proprietary software industry, though the Open Source
community, as a resistant/would-be-legitimizing identity is also a position against which the
Free Software community struggles. Recall from my discussion in the previous chapter that
the proprietary software industry evangelizes an entrepreneurial trajectory for software,
programming, and the software programming culture. The Free Software Community, on the
other hand, evangelizes an idealistic trajectory, in which sharing is valued more than
economic profit, morality more than capitalism, and unlimited democratic freedom more than
democratic freedom defined by the producer/consumer relationship. Because this idealistic
trajectory runs counter to the dominant entrepreneurial trajectory and also seeks to abolish
the practices that limit the rights of users to share free software, the Free Software
community is positioned defensively in a struggle against the proprietary software industry,
specifically, and, as a collective reconfiguration of the resistant/would-be-project identity,
against the legitimizing identity, generally. This struggle of the Free Software community is
evident in the GNU GPL, a software licensing agreement written by the FSF in 1989.

Like Microsoft’s EULA, the GNU GPL is structured by the generic features of
copying and distribution common to software licensing agreements as application of
copyright. However, unlike Microsoft’s EULA, the application of copyright in the GNU GPL
does not deny licensees the right to copy and distribute GPL software in order to maximize
the amount of revenue generated from the software. As stated in Section 3 of the GPL, “You
[the licensee] may copy and distribute the Program…in object code or executable form”38. In
giving licensees the right to copy and distribute software, the GPL legally constructs software
as free and inverts the EULA application of copyright. Whereas the EULA denies licensees
the right to copy and distribute proprietarily-owned software for the sake of protecting the
rights of the producer/inventor, an application of copyright that had become the dominant in
the software programming culture, the GPL shares with users the rights given to the
copyright-holder by copyright, a process that the FSF coined copyleft.

---

38 Executable form is software translated by a compiler (a program that does this translation) from the source code written in
a programming language into object or binary code that is then readable by the hardware.
However, the GPL’s copyleft application of the copying and distribution features of software licensing agreements does not mean the GPL is without any limitations to these features. The primary limitation imposed by the GPL upon a licensee is the denial of the right to limit the rights of others, a strategy that prevents free software from being co-opted by those who would make the software proprietary. Instead, the GPL insures that every licensee is given through copyleft the same freedoms ensured by copyright for the copyright-holder. As stated in Section 6 of the GPL, “Each time you redistribute the Program…the recipient automatically receives a license from the original licensor to copy [or] distribute… the Program subject to these terms and conditions.” Because copyleft legally dictates that licensees have the same rights as copyright holders, the denial of rights is also the same. Licensees are denied the right to limit the rights of others just as are the licensors. To insure this denial, the GPL’s Section 4 stipulates that rights given to a licensee “will be terminated” should the licensee alter the GPL’s copyleft application of the generic features of the GPL. This limitation insures that everyone, whether licensor or licensee, has the right to copy and distribute freely shareable software. Consequently, we can understand this application of the generic features of copying and distribution as an inversion of the most common applications, such as those found in Microsoft’s EULA. The GPL is, thus, a legal technique that insures the privileging of morality over profitability as imagined in the Free Software community’s idealistic trajectory. As explained in the GNU’s preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This “preamble,” a feature unique to the GPL, functions so as to differentiate the GPL from the standard proprietary software licensing agreements that have governed the majority of software use since the 1980s. These proprietary software licensing agreements, such as the EULA, had led to the unprecedented generation of revenue by proprietary software vendors, who as licensors denied consumers the freedom to copy proprietary software so as to insure the freedom of producers to sell software for an economic profit generated by the sale value of the software, a freedom guaranteed by copyright. Copyleft, on the other hand, allows
anyone, not only the licensor, to copy and distribute GPL software, and, should a licensee choose, to economically profit from copyleft: “You may charge a fee for the physical act of transferring a copy…” More importantly for the idealistic construction of software and programming, the GPL, through the copyleft application of the generic features of copying and distribution, legitimizes software as a freely shareable resource and challenges the dominant representation of software as a commodity whose economic value necessitates protection. Because the value of the software is not located in the sale value of the product, the license does not need to protect the rights of the copyright-holder to earn capital from the commodity of software by limiting a licensee’s right to copy to one back-up copy and to deny completely the right to distribute the software. Thus, as necessitated by the Kantian-like idealism of Stallman, the FSF, and the Free Software community as a whole, the GPL empowers both the licensor and the licensee with the democratic freedom to share, a right that is constructed as a moral-imperative in the idealistic trajectory of the culture and as counter to the entrenchment of dominant society in the intellectual property-as-commodity mentality.

Another GPL limitation that insures the application of copyright as copyleft is that redistributed copies of GPL software must “give any other recipients of the Program a copy of this License along with the Program” so that subsequent licensees understand that they have the unalienable, moral right to share software, just as imagined in the idealistic trajectory reproduced and reified by the Free Software community as a resistant/would-be-project identity. Mandating that a copy of the GPL must accompany any legally copied and distributed GPL program insures that licensees understand the rights and limitations given to them as licensed users. This mandate also insures that a licensor does not add any other limitations to those already stipulated in the licensing agreement. As the second part of Section 6 states, “You [the licensee] may not impose any further restrictions on the recipients’ exercise of the rights granted herein.” Of primary importance to the success of the GPL is that a licensee understands that a marriage, to borrow a Microsoft metaphor, exists between the GPL and the free software it licenses. However, unlike Microsoft, a marriage does not exist between the licensed program and a piece of hardware, not the computer on which the licensee first installs the program or the computer upon which an OEM installed it.
The aim of the GPL is not to maximize for the economic profit through the sale value of the program; this is an aim of the entrepreneurial construction of software. Rather the aim of the GPL, through the copyleft application of the copying and distribution features of the software licensing agreement genre, is to maximize the number of users who have access to and can subsequently share the program. Consequently, we can understand the GPL’s application of copyright through copyleft as a rejection of the copying and distribution limitations imposed by proprietary software licensing agreements by giving licensees the right to copy and distribute GPL software. However, this right is not limited by the GPL to executable software only. The GPL also extends these rights to source code.

Rights and limitations regarding source code are never discussed in the EULA because Microsoft, as a proprietary software vendor, black-boxes its source code, an important technique for protecting software’s sale value, itself a rhetorical construction of the entrepreneurial narrative and its supporters in the proprietary software industry. The GPL, on the other hand, challenges this construction and legitimates source code as freely shareable. Consequently, the GPL neither restricts the number of copies of source code that can be made from a licensed copy, nor limits the right to distribute copies of a program’s source code. However, as with the copying and distribution limitations of the actual executable program, the primary limitation to this source code-related right is that licensees are not permitted the right to prevent others from having this right. The right to share extends to the copying and distributing of source code. As such, the GPL’s copyleft application of copyright also describes the copying and distribution features for source code. As stated in Section 1 of the GPL,

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice…keep intact all the notices that refer to this License…and give any other recipients of the Program a copy of this License along with the Program.

By extending the copying and distribution applications to source code, the GPL insures that copyleft applies to the tool by which licensees can understand how an executable program works. The GPL gives not only gives the right to share source code freely but
mandates through copyleft that every licensee must be given the same right to share source code as the copyright holder. Just as with the limitations regarding the copying and distribution of a program, the same limitations to source code insure that a licensee can not withhold from others the same rights s/he has enjoyed because of the GPL because these rights are transferable. Should licensees illegally attempt to subvert the rights and limitations regarding source code, the GPL’s Section 4, which also covers the copying and distribution of program, terminates the right of the former licensee.

The GPL’s application of the generic features of copying and distribution make material the idealistic trajectory espoused by the Free Software community. Through the GPL, sharing becomes a legitimate practice protected by copyright through the application of copyleft. The idealistically-driven right to share software under the moral imperative of the golden rule is rooted in copyright law by the OSD’s inversion of copyright through copyleft. The deliberative application of software licensing agreements’ generic features of copying and distribution ensures that the Free Software community can treat software and its related practices as a means to follow the idealistic trajectory that values humanity more than profit. This belief necessarily positions the Free Software community defensively, as a reconfiguration of the resistant/would-be-project identity, against the capitalistic, profit-motive of the proprietary software industry as a reconfiguration of the legitimizing identity that rationalizes the dominant institutions of society.

**OSD**

The OSD is a meta-license and consequently differs from the EULA and GPL in certain ways, though the OSD is still structured by the generic conventions of software licensing agreements. In the words of the Open Source Initiative (OSI), the OSD “spells out the essential qualities of open-source software” and how these qualities should be applied in software licensing agreements that regulate the use of open source software. Specifically, the OSD specifies what constitutes open-source applications of the generic features of software licensing agreement, including copying and distribution. As the “Introduction” to the OSD explains, “Open source doesn’t just mean access to the source code. The distribution terms of open-source software must comply with the following criteria….” By creating software licensing agreements that comply with the “essential qualities” outlined in the OSD, a
licensing agreement and the software it licenses can be OSI-certified. To show what software has been licensed in accordance with the OSD and the values of the Open Source community, the OSI created a certification mark (see Figure 6.2)—“OSI-certified”—and an identifying graphic to affix to the medium the software is delivered in.

Figure 6.2: The OSI’s Certification Mark for OSD-Compliant Software

Marking software with either the certification mark or graphic is intended to inform licensees that, in the words of the OSI, “the software is being distributed under a license that conforms to the Open Source Definition. Use of these marks for software that is not distributed under an OSI approved license is an infringement of OSI’s certification marks and is against the law.” To receive the certification mark or graphic, a software licensing agreement must be approved by the OSI.

In the “essential qualities” relating to the generic features of copying and distribution, the OSD is very similar to the GPL—not surprising given that the GPL is OSI-certified and that the OSI is, in the words of its author, a “derivative” of the GPL. Like the GPL, but unlike the EULA, the OSD specifies that open-source licensing agreements must allow, according to Section 1, “free redistribution.” Open-source licensing agreements, in order to be OSI-certified, must allow licensees to make copies of the software and to redistribute those copies to others. This is the application of the generic features of copying and distribution that allow for the sharing of tools and that allow this community, in the spirit of scientific inquiry, to test these tools in a variety of hardware/software configurations. According to the pragmatic narrative that the Open Source community espouses, only through a process such as this can the most useful tools be developed for the practical needs of programmers. Section 1 of the OSD ensures that any open-source licensing agreements certified by the OSI will mandate free redistribution, and hence, free copying of the software.
being licensed, thereby making material the values of the pragmatic trajectory through the open-source application of the iterative regulatory technique of copyright.

In addition, the OSD specifies in Section 1 that “any license shall not restrict any party from selling or giving away the software…” Because the Open Source community, as a reconfiguration of a resistant/would-be-legitimizing identity, embraces commercialization not of software as a product but of the services related to software, the OSD mandates that OSI-certified licensing agreements must allow the software it licenses to be used for any purpose—commercial or otherwise. Even when such software is included in “an aggregate software distribution from several different sources,” including proprietary sources, OSD-compliant licenses “shall not require a royalty or other fee.” Though software licensed through an OSD-compliant licensing agreement could be bundled with proprietary software, the licensing for which demand a fee for use of the proprietary software, the OSI-certified license for open source software could not demand a similar fee that, as in the case of Microsoft’s EULA, would possibly necessitate a “refund.” As stated in Section 9 of the OSD, “The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.” In Version 1.0 of the OSD, this stipulation used the word “contaminate” to describe what an OSD-certified licensing agreement must not do to other programs. Having revised the OSD in Version 1.9 to eliminate the negative connotation of “contaminate” and instead stresses that an open-source licensing agreement cannot take away the choice of licensees to bundle open-source software with proprietary software. Though the GPL, which is OSI-certified, complies with Section 9, the GPL embeds this right. By creating a section dedicated to this right, the OSD explains to those who might be put off by the concept of free software that free software in no way undermines, or contaminates, other licensing agreements, a fact not made clear by the GPL.

Like the GPL, but unlike the EULA, the OSD mandates that any OSI-certified licensing agreements must stipulate that “the program must include source code, and must allow distribution in source code as well as compiled form.” Because the right to copy and distribute software only allows licensees to run and test software, the ability for a scientifically-motivated community, such as the Open Source community, to understand why
the program does what it does is pivotal. Only by reading the source code of a program can a programmer comprehend how the program works. Consequently, Section 2 mandates that OSI-certified licensing agreements must give the freedom to share source code. However, this mandate must not be confused with the same mandate given in the GPL. The sharing of source code for the Open Source community is pragmatic, a right necessary for the development of the most useful software, not for the morally-driven idealism of the golden rule as with the GPL. To emphasize this difference between the GPL and the OSD, the OSD mandates in Section 6 “No Discrimination Against Fields of Endeavor.” Though the GPL has no overtly anti-business rhetoric, the narratives espoused by the Free Software community, in its defensive position against the dominance of the proprietary software industry, has vilified this industry with its rhetoric that proprietary software denies users freedom (see Chapter Five). To distinguish itself from the Kantian-inspired idealism of the Free Software community, the OSI, as a flagship organization in the Open Source community, has typically evangelized, along with other members of the community, the commercial-friendliness of open source, though one that is different from the dominant mode of entrepreneurialism that constructs software as a commodity. Legitimizing the position of the Open Source community as a resistant/would-be-legitimizing identity, the OSD stipulates, “the license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.” The Open Source community is in no way anti-business and Section 6 of the OSD is meant to testify to this, just as the move away from “free software” to “open source,” which I discussed in Chapter Five, was. As the OSI explains in a rationale to this section, “We want commercial users to join our community, not feel excluded from it.”

Consequently, in the application of the copying and distribution features regarding software and its source code, the OSD actively position itself against both the proprietary software industries way of doing business, but not business itself, as well as against the idealism of the Free Software community. Instead, the OSD forwards the pragmatic trajectory of the Open Source community, a trajectory that values choice and the scientific creation of knowledge over the dogma created through capitalism and idealism. In this way, the OSD solidifies the pragmatic trajectory and the position of the Open Source Community
as the embodiment of the resistant/would-be-legitimizing identity in the software programming culture.

**Modification**

Modification is another generic feature of licensing agreements. The modification feature specifies the rights and limitations given to a licensee to modify a program’s source code, the programming language in which a program is written. Access to a program’s source code allows anyone literate in the language to understand how a program works and, should the right be given by the licensing agreement, modify the source code. Microsoft’s EULA makes no provision for the modification of source code as the proprietary software vendor does not typically make the source code for its products available to users. The GPL and the OSD, on the other hand, create the opportunity for modification as these licensing agreements specify that source code must accompany any software licensed under these agreements. In creating the opportunity for modification, both the GPL and OSD take the next step—the former for idealistic reasons, the latter for practical reasons—and give licensees the right to modify that source code and, in keeping with the rights regarding distribution of software and source code, to distribute those modifications as derivative works.

**Microsoft’s EULA**

In the proprietary software industry, source code is the source of a program’s economic value, as I explained in Chapters Four and Five; consequently, the proprietary software industry and the entrepreneurial narrative it espouses construct source code as a resource to be protected. This protection has typically occurred through the black-boxing of a program’s source code. By not providing the source code for its products, Microsoft, as well as other proprietary software vendors, protects the economic value that is accrued from the sale value of its black-boxed software. Consequently, Microsoft does not make any provisions in the EULA for the modification of its software because such an act is impossible without the source code, itself. Because transparent source code allows others other than the developer to read the knowledge work behind the program, this threatens the democratic habits of capitalism in which producers accrue capital by laboring to create a commodity for
those who have an unfulfilled need. Through the EULA, Microsoft’s products licensed through this application of copyright are legitimated as a resource not to be licensed even with the strictest limitations. Should everyone have access to a Microsoft program’s source code, anyone would be able to use that code as a resource without the R&D investment that allowed that source code to be written in the first place, a process that goes against the entrepreneurial trajectory of the proprietary software industry.

**GNU GPL**

Unlike Microsoft’s EULA, which makes no provision for the modification of a EULA source code, the GNU GPL allows for the modification of the source code of any GPL software. Because the copying and distribution generic features of the GPL stipulate that source code must be made available, the modification of source code is made possible and thus necessitates regulation by the GPL. Consequently, Section 2 of the GPL stipulates that the right to modify source code is given to a licensee: “You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program…”

Recall from my discussion in Chapter Four that a motivating factor in Stallman’s decision to begin the GNU Project was not only the denial by a proprietary copyright holder the right to read the source code for a program but the denial to modify the source code so that the printer-related program would function as Stallman and others need it to. Because of the economically-driven choice to deny both access to source code and the modification of source code, Stallman began to evangelize the idealistic trajectory for the object of software, the practice of programming, and the software programming culture so that the practice of sharing would once again become a moral imperative that united people rather than dividing them. Section 2 of the GPL legitimates in copyright law the right to modify software as an idealistically-motivated practice that, in the words of Stallman, “values humanity” more than anything else.

Additionally, in keeping with its copyleft distribution feature, the GPL also allows for modified source code and the derivative program created from it to be distributed; as Section 2, entitled “Source Code,” further explains, “you may…copy and distribute such modifications or work under the terms of Section 1 above….” However, in order to distribute modifications, licensees must do the following: provide “prominent notices” that state that
the files have been changed and the date of the change, license the derivative work under the GPL, and have the program, if it is interactive, display or print specified information including the copyright. Commonly referred to as the “viral” feature of the GPL, it mandates that the GPL must be the licensing agreement that regulates the copying distribution, modification, and warranty of derivative works resulting from GPL software. Through this viral attachment of the GPL to derivative works, the GPL insures that the propagation of future GPL software, as well as the idealism that the GPL legitimates in the material use of software supported by the Free Software community.

**OSD**

Just as the GPL is a regulatory technique used by the Free Software community to legitimate free software, the OSD is a technique used by the Open Source community to legitimate open-source software. Whereas the GPL sets the conditions by which programmers can exercise their democratic freedom to act according to their moral principles, the OSD sets the conditions by which programmers can exercise their democratic freedom to choose the most useful software for their needs. To extend the freedom to choose, the OSD stipulates that OSI-certified licensing agreements must mandate that source code is modifiable and that these modifications are, in keeping with the distribution application of software and source code, redistributable as derivative works. In fact, Section 2 of the OSD, which stipulates that open-source licensing agreements must make source code transparent and distributable, states “the source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed.” The reason to include transparent source code is to make that source code modifiable. Should a programmer have a need to modify a program to fit his needs, the OSD ensures that she will have the right to do so with software licensing by an OSI-certified application of copyright. Without the ability to modify software, the usefulness of a scientific model of software programming would be extremely limited. To create knowledge, programmers must have the ability to make knowledge by building on the work of other programmers. According to the pragmatic narrative, this process increases the likelihood of creating the most effective software.
Because this scientific model of programming is dependent upon the work of the community, the OSD mandates that OSI-certified licensing agreements must make the distribution of derivative works created from open-source software possible. Section 3 of the OSD—“Derived Works”—explains, “The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.” Because the running of a program in executable form shows what the program does or does not do, and because the reading of the source code reveals why the program does what it does or does not do, the next possible step in creating the most effective software is modification, should software testing reveal a need for modification. Because hardware and software configurations create an infinite number of environments in which a particular program runs, the greater the number of programmers running and testing the program, the greater the likelihood that the usability, stability, and security—the quality of the software—will increase, according to the Deweyian-inspired pragmatic narrative of the Open Source community. Through the OSD, the OSI and other members of the Open Source community instantiate the conditions by which the pragmatic trajectory and its scientific model for software programming is made material.

**Warranty**

The final generic feature of software licensing agreements is the warranty feature. A warranty lays out the responsibility of the manufacturer to the consumer in regards to the purchase of a product. Traditionally, the warranty represents the manufacturer ensuring, or standing behind, the quality of the product. Though a number of warranties exist (e.g., the express warranty, the extended warranty, the limited warranty, the merchantability warranty, the functional warranty, etc.), a warranty is not always attached to software licensing agreements; in these cases, a warranty disclaimer may be included in the agreement. In my analysis, the only licensing agreement that offers a warranty is EULA. The GPL offers a warranty disclaimer and the OSD makes no specifications regarding warranties but instead leaves the issue of providing or disclaiming warranties to the distributors. I argue that the choice to offer a warranty, to offer a warranty disclaimer, or to take no position on warranties is reflective of the spatial positions of the proprietary software vendor Microsoft, the Free Software community and the Open Source community.
Microsoft’s EULA

Providing a warranty, a guarantee of some kind between the manufacturer and the consumer, is a common feature associated with the purchase of a commodity. As explained by the U.S. Federal Trade Commission, “When you make a major purchase, the manufacturer or seller makes an important promise to stand behind the product. It's called a warranty.” In my analysis, the EULA is unique in its offer of a warranty; though, given its application of copyright by a proprietary software vendor that generates revenue from the sale value of software as a commodity, this is not surprising. Microsoft, as a corporation, is able to stand behind its product in a way that diffused communities of individual software developers in the Free and Open Source community are not necessarily able. A COA gives a licensee of EULA software a limited warranty by Microsoft:

LIMITED WARRANTY. Manufacturer warrants that (a) the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Microsoft hardware accompanying the SOFTWARE will be free from defects in materials and workmanship under normal use and service for a period of one (1) year from the date of receipt. Any implied warranties on the SOFTWARE and Microsoft hardware are limited to ninety (90) days and one (1) year, respectively. Some states/jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

The function of this limited warranty, which actually stipulates the duration of two different express warranties, is to demonstrate to the consumer that the manufacturer stands behind and guarantees its product’s performance. The first express warranty, explained in “(a),” guarantees for 90 days the functionality of the Microsoft software commodity; the second express warranty, which is also a function warranty and is explained in “(b),” guarantees for one year the functioning of the hardware the software is delivered upon. By providing the limited express warranties through the EULA, Microsoft, following a long-tradition in commodity manufacturing, represents for licensees Microsoft’s belief in the quality of its product. As I discussed in my analysis of the entrepreneurial narrative of the proprietary software industry as a legitimizing identity, Microsoft, from the earliest archived narrative of
its co-founder Gates in 1976 to the 21st-century story-telling of proprietary software evangelists, believes that the best software is created by professional programmers employed by corporate proprietary software vendors. By providing in the EULA warranties that insure to the consumer the economic worth, and hence, quality of the software, Microsoft utilizes the warranty strategy of industrial capitalism. By standing behind a product, the manufacturer reassures the consumer that the commodity will function as it should, or consumers, having paid for the use of the software, can receive a refund for the capital spent on the product within an amount of time specified by the warranty.

**GPL**

Though warranties are common in proprietary software licensing agreements, such as the EULA, free software typically does not offer a warranty. Because free software does not have a sale value, there is no refund to be given. But this does not mean that free software lacks a protection automatically given through proprietary software. Instead, licensees have the right to discontinue the use of one free software program for another at any time without having to pay for the software itself. For instance, the GPL offers a warranty disclaimer, not a warranty. As stated in the GPL,

```
BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW EXCEPT WHEN OTHERWISE STATED IN WRITING THE
COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE
PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER
EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND
PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL
NECESSARY SERVICING, REPAIR OR CORRECTION.
```

Because GPL software is contributed by programmers who are motivated to do so ideologically, rather than sold by a corporation that can “stand behind” its product, the disclaimer of warranty functions to protect those who make these contributions. The Linux
kernel and much of the GNU Project, both of which are licensed under the GPL, are the result of the freely contributed software programming of those who do not necessarily align themselves with the entrepreneurial narrative of the proprietary software industry. To make these programmers liable through the inclusion of a warranty is the GPL would possibly deter programmers, even if they were ideologically inclined, from making these contributions of free software. Consequently, GPL software is given “as is,” with the risk taken by users, a fact whose rhetoric alone can dissuade users, particularly lay persons, from embracing free software. However, given the limited nature of the warranty offered by licensing agreements such as the EULA, software use is, in fact, always a risk proposition for users.

**OSD**

The OSD makes no specifications about warranties. However, the lack of any specification, with either a warranty given or a disclaimer of warranty, allows OSI-certified licensing agreements to decide how this feature of the generic structure of licensing agreements will be applied. Distributors of open source are welcome to offer a warranty, and many Linux distributors do so. At the same time the GPL, which as I have stated is OSI-certified, is free to make a disclaimer of warranty. As with the applications of the generic features of copying, distribution, and modification, the OSD is intent on maximizing licensors ability to choose what works best in a software-related situation. Consequently, the lack of specifications regarding warranties is in keeping with the pragmatic trajectory that values the democratic right to choose over all else. If a Linux distributor, for example, chooses to offer a warranty, this is a service for which the distributor can charge a fee, thereby increasing the means by which a successful service-orientated open-source software company can succeed. This is a right also given by the GPL, though, with its lack of emphasis on the ways in which businesses can be formed around free software, is never acknowledged. Consequently, the GPL’s disclaimer of warranties can appear to deny the offering of warranty in connection with GPL software. The OSD avoids this problem by making no mention of warranties. The practicalities of warranty offers are left to the practical needs of licensees, as necessitated by the pragmatic trajectory of the Open-Source community.
I argue that these different applications of the generic features of software licensing agreements—the copying, distribution, modification, and warranty features—are a result of the deliberate action of the licensors who choose to license their software either under licensing agreements, such as the EULA or GPL, or under licenses that conform to the regulations specified in the OSD. As illustrated in Figure 6.1, these licensing agreements are the result of practical evaluation, mediated by the temporal orientations of iteration and projectivity as well as by the spatial positions organized around them.

Copyright, as a regulatory practice, is structured as a typified response through the genre of licensing agreements and the need to regulate software use. However, copyright is also a resource through which the ideological trajectories, which result from the reconfiguration of the competing identities of the network society by actors in their roles specific to the software programming culture, can be made material and regulated by law.

This analysis of the typical licensing agreements used by the proprietary software industry and the Free Software community, as well as the meta-license that regulates the construction and adoption of licensing agreements compliant with those “essential qualities” that constitute open-source software, further illustrates the network of alliances between various communities in the software program culture. The GPL, though a result of the interplay between the structure iterative practice of copyright and the Kantian-like idealism espoused by the Free Software community, is OSI-certified. The use of the GPL allows for the same everyday practices as advocated by the OSD. However, the use of the GPL is problematic for the Open Source community, given its pragmatic ideology. The language of the GPL, especially its preamble that begins by setting-up the “freedom” of the GPL in opposition to dominant proprietary software licensing agreements that “take away your freedom,” is rooted in the often explicitly anti-business rhetoric of the idealistic narratives of the Free Software community. The OSD aims to ideologically separate itself from the ideologically-motivated rhetorical construction of the GPL by emphasizing, especially to the business world, the practical benefits of OSD-compliant licensing agreements. Consequently, the OSD is an application of copyright that seeks to align the Open Source community with the ideology, but not the practices, of the proprietary software industry, specifically, and the business world, generally.
Testament to the OSD’s success in this area is Microsoft’s Shared Source Initiative. The philosophy behind this initiative is the “balanced approach that allows [Microsoft] to share source code with customers and partners while maintaining the intellectual property rights needed to support a strong software business” (Microsoft, Shared Source Philosophy). The licensing agreements used to regulate this balanced approach were from the beginning little more than what has been commonly described as “look but don’t touch” specifications that gave licensees—typically government, business, or academic partners—the right to view a copy of a Microsoft product’s source code. The right to modify that source code, a pivotal feature to both the GPL and OSD, was not given to users. However, in October 2005, Microsoft released three new Shared Source Licensing Agreements to replace the ten licenses that had been in use since its Shared Source Initiative began in 2001. Two of the three replacement licenses—the Microsoft Permissive License and the Microsoft Community License—allow transparent source code to be modified, and given the specifications of the OSD, could possibly be OSI-certified should Microsoft apply for certification. As of March 2006, the only programs to be released under these modification-granting licenses are starter kits from Visual Studio 2005. Nevertheless, the fact that the bastion of proprietary software industry has created, and is in some small way using, an OSD-compliant licensing agreement for its formerly proprietary software is testament to the OSD’s success in distinguishing itself from the ideology of the Free Software community—with its resistant/would-be-project identity, which seeks to overturn the dominant structures of society—and from the ideology of the proprietary software industry and its legitimizing identity that rationalizes the dominant structure.

**ENFORCING SOFTWARE LICENSING COMPLIANCE**

Because the EULA and GPL software licensing agreements use copyright law and because the OSD makes specifications for open-source licensing agreements that also use copyright law, the legal and illegal use of these licensing agreements create the need for enforcement. Though the entrepreneurial narrative is the only construction, thus far, to make explicit the need to police and enforce the use of software commodities, the GPL and OSD have also necessitated the enforcement of the rights and limitations given to its licensees in order to protect GPL- and OSD-regulated software from being appropriated for black-boxed
proprietary software. In fact, both the GPL and OSD are legal techniques by which to prevent the disruptive software technology of free and open-source software from being used in software that is not licensed so as to allow the sharing of software and source code. With the increasing use of free and open-source software, the need to ensure “compliance” with the licensing agreements that regulate this software has also increased.

For example, though the products of the proprietary software industry still comprise the majority of the software in use today, the corporate sponsorships and partnerships, along with their investment in free and open-source software development, have helped to make disruptive software technology a real choice for many users, though these choices tend to erase the idealistic trajectory of free software. (IBM’s marketing campaign that advertised the usefulness of “open source” through the campaign’s motto—“Linux is growing. The future is open—certainly helped this endeavor.) Perhaps the most powerful example of the growing success of disruptive software technology is the move by governments looking to “break free of the United States’ lock on the global software market” (Festa, 2001). Microsoft, who has supplied throughout the globe endless numbers of government, educational, and individual users with established software (to the point of industry dominance), is being challenged to a certain extent by GPL-regulated software and OSD-modeled software licensing agreements. In 2000, governments in Europe, at both the local and federal levels, spent approximately $7.8 billion on software. To counteract spending such as this, federal governments in countries such as Brazil, Peru, and China have turned increasingly to open source software. Peru has even passed legislation, “Free Software in Public Administration,” which decrees that open-source software must be used in all government systems. Japan, China, and South Korea are currently collaborating on open-source alternatives in an effort to diminish their use of Microsoft products as well (The Economist, 2003). And city governments in Munich, Germany, and Florence, Italy, recently dumped Microsoft in favor of Linux and other open source applications.

Even in the U.S., state governments in Texas—State Bill 1579—and Oregon—House Bill 2892—have attempted to pass legislation calling for the use of open source software. And in Massachusetts, the state government has created the “Enterprise Open Standard Policy,” which requires that all state-funded agencies must “consider all possible
alternatives—proprietary, open source, and public sector code sharing—in determining best value solutions” (Commonwealth of Massachusetts, 2004). The adoption of disruptive software technology by these governmental institutions represents the growing success of disruptive software technology. With the increased adoption of free and open-source software, though in the case of free software not the ideology that encodes it, the licensing agreements that regulate the use of this software have needed enforcement.

Enforcing the EULA

Enforcing proprietary software licensing agreements, such as the EULA, in order to protect the sale value of the commodities, has entailed a number of actions of strategies. Programs licensed through the EULA include software technology that terminates after 30 days a licensee’s use of the software should the licensee fail to “supply information required to activate your licensed copy in the manner described during the setup sequence of the SOFTWARE. (EULA).” As stated in the EULA,

There are technological measures in this SOFTWARE that are designed to prevent unlicensed use of the SOFTWARE. Microsoft will use those measures to confirm you have a legally licensed copy of the SOFTWARE. If you are not using a licensed copy of the SOFTWARE, you are not allowed to install the SOFTWARE or future SOFTWARE updates. (EULA)

The creation of an activation process that requires information from the Microsoft product’s COA allows Microsoft to exercise surveillance over its legally licensed copies. And the surveillance technology included in the software purchased for another function allows Microsoft to terminate the rights of licensees who have not provided the information stipulated as a legal requirement for the use the software.

Additionally, the dominant structure of society provides the means by which to enforce and, should it be necessary, punish illegal piracy of proprietary software. Individual violations of software copyright by individuals who, though licensed to use only one copy of software on one’s own computer, installs that software on multiple machines within one’s own home, is difficult to identify and hence prosecute, a fact that led Gates in “An Open Letter to Hobbyists” to call for the policing of hobbyists by hobbyists. Piracy involving the large-scale copying and distribution of proprietary software occupies not only the software
watch-dog groups that police software use on the behalf of the proprietary software industry (see Chapter Five) but also U.S. government agencies, most notably the Computer Crime and Intellectual Property Section (CCIPS) of the Criminal Division of the U.S. Department of Justice (U.S. DOJ). The CCIPS’s primary task is the enforcement of Article I, Section 8 of the U.S. Constitution as it relates to software and digital content in Acts such as the No Electronic Theft Act of 1997.

The No Electronic Theft Act of 1997 specifies as a “criminal offense” the infringement on copyright “through the reproduction or distribution, including by electronic means, during any 180-day period, of 1 or more copies or phonorecords of 1 or more copyrighted works....” Penalties, including imprisonment and fines, are determined by the number of copies and “retail value” of the works that are pirated. Under this Act, a number of cases have been prosecuted on behalf of proprietary software vendors. The first software-related case prosecuted under this Act was U.S. v. Levy. In November 1999, through the “cooperative efforts of the FBI, the Oregon State Police, the University of Oregon, the Software Information Industry [sic], the Business Software Alliance...” and others (U.S. DOJ), Jeffrey Levy, a senior at the University of Oregon, was prosecuted for illegally copying and distributing software and content pirated from the proprietary software industry, as well as other industries that rely on copyright protection of digital content (e.g., the music and entertainment industries). Because the retail value of the software and other copyrighted material (e.g., digital movies and music) posted on his website could not be determined at the time, Levy’s punishment was only two years conditional probation.

As a result of the conditions created by proprietary software licensing agreements, such as the EULA, and reinforced by U.S. law, Dashiell Ponce de Leon was sentenced in October, 2005, to 46 months in prison and ordered to pay approximately $1.15 million in restitution for copying and distributing over $1 million in pirated software and videogames via his websites. Proprietary software vendors who had their copyrights infringed upon included Adobe Systems, Inc.; Macromedia, Inc.; and Microsoft Corporation. According to the press release issued by the U.S. DOJ, the sentence was intended to deter others from software copyright infringement: “Focusing on the harm done to the copyright industries, in imposing sentence, the court noted that the sentence in this case was warranted based upon
the need for adequate deterrence of this defendant and others who might be tempted to do as he had done and also the protection of the public.”

But copyright infringement is a two-way street. Just as illegal reproducers and distributors of proprietary software are guilty of a crime, so too are those who use illegally reproduced and distributed software. Microsoft’s EULA, as well as other proprietary software licensing agreements, states that the software must be accompanied with a COA affixed to the software packaging or computer that the software is already installed on. Without those, the software is not authentic, and consequently, no legal licensing agreement is in place. Though individuals are hard to prosecute because of the privacy of use in the home, many organizations have been caught using software that violates copyright. For instance, in February 1999, the Los Angeles Unified School District agreed to pay the BSA, $300,000 for unlicensed copies of proprietary software, including Adobe, Microsoft, Novell, and Symantec, all of which were being used on school computers. The BSA was alerted to the school district’s copyright infringement after a call to the watch-dog group hotline: 1.800.NO PIRACY.

These acts of enforcement of proprietary software licensing agreements such as Microsoft’s EULA are a seemingly natural consequence of copyright law. If someone violates the copyright of a licensor, then that person is subject to certain punishments. However, as the lack of enforcement in the GPL illustrates, such punishments are a choice, not a “natural” consequence of copyright.

Compliance and the GNU GPL

Though the Free Software community has not utilized the same enforcement techniques that the proprietary software industry has—not surprising given this community’s resistance to the dominant structures of society as a resistance/would-project-identity—this community is nonetheless dedicated to ensuring “compliance” with the rights and limitation specified in the GPL. This dedication is a result of the Free Software community’s desire to prevent the appropriation of free software by those who in developing proprietary, black-boxed software deny users the freedom to share. Most notably, the Free Software Foundation has created the “Free Software Compliance Lab,” the evolution of which is described on the Foundation’s website:
The GPL Compliance Lab has been an informal activity of the Free Software Foundation since 1992 and was formalized in late 2003. The Compliance Lab is our department handling the investigation of GPL...violations and subsequent enforcement when violations are confirmed. The Lab also assists other copyright holders (besides FSF itself) when they seek to enforce the GPL. Finally, the Lab provides general "knowledge infrastructure" concerning the GNU GPL and Free Software licensing; we answer many licensing questions from the public and from lawyers working in the field each day.”

To facilitate the enforcement of the GPL, as well as to help those who might want to adopt the GPL for their software, the Lab assists in investigations of abuses of the GPL and provides information so that illegal users can become compliant.

To date no lawsuits have been filed for copyright infringement on GPL-licensed software. According to Eben Moglen (2001), General Counsel for the Free Software Foundation, “Our position has always been that compliance with the license, and security for future good behavior, are the most important goals. We have done everything to make it easy for violators to comply, and we have offered oblivion [forgiveness] with respect to past faults.” By discouraging enforcement and encouraging compliance, the Free Software Foundation, the flagship organization for the Free Software community, has resisted punishing, as is its right under copyright law, violators of the GPL.

**Compliance and the OSD**

Like the Free Software community, the Open Source community is more concerned with compliance than punishment. A number of open-source related businesses have offered compliance consultation to help businesses correctly use software in the ways compliant with the generic features of software licensing agreements modeled after the rights and limitations described in the OSD. For instance, Tangient, an open-source consulting business, offers assistance in ensuring compliance, especially given that those who use proprietary software licensing agreements, such as the EULA, may be confused about the correct use of OSI-certified open-source software. As their website states, “You think you’re in compliance with these open source licenses but you’re not sure. You’d like to protect your IP and mitigate risk but you don’t want to spend a lot of time or money” (“Get Started”). Once
again, compliance rather than punishment is the objective. Though the Open Source community wants licensees to follow the rights and limitations dictated in OSD-compliant software licensing agreements, punishing individuals and organizations for violations of these agreements is not considered useful because it stands to turn individuals and organizations off from using the disruptive software technology of open source.

This genre analysis of Microsoft’s EULA, the Free Software Foundation’s GNU GPL and the Open Source Initiative’s OSD illustrates the ways in which a structured iterative practice of the software programming culture, even one that is institutionalized by dominant society, can be reconfigured so as to achieve different ideological trajectories. In analyzing how software licensing agreements, as a form of copyright, are “both means and end, both resource and product” (Miller, 1994, p. 70), I sought to explore the ways in structured practices provide stability, while, at the same time, provide the means by which to enact change. Through the different applications of software licensing agreements, the regulatory practice itself was reproduced. However, the sets of relations created through the discursive construction of ideological trajectories of possible action were also reproduced. In fact, through the regulatory practice of copyright, these social positionings were cemented in law.

Additionally, the examination of the ways in which copyright is or is not enforced by the communities legally constituted by these different licensing agreements and meta-license illustrates how the idea of “punishment” can be re-envisioned. It is certainly not surprising that these re-envisioning of punishment from enforcement to compliance occurs through the discourse of those communities that embrace a shared resistance identity, even though those identities may not be exactly the same.

For the lay user of software, the genre analysis that culminated from the context-building facilitated by the spatio-temporal framework is intended to illustrate the ways in which software use, even by lay users, is discursively coded ideologically. The result of this coding is that lay users are subject to ideologies that may or may not be aligned with their own internalization of identity. In the concluding chapter of this dissertation, I offer possible avenues by which lay users of software can reflectively participate in the discursive coding of software.
CHAPTER SEVEN: TOWARD A DIGITAL LITERACY

In this concluding chapter, I seek to answer my final research question: What are the means by which lay persons can develop a digital literacy of software. Though I have hinted at possible ways to answer this question throughout this dissertation, I would like to develop in this last chapter a more explicit, though still partial, answer about how lay persons can develop digital literacy of software. Answering this question was facilitated by the analysis I undertook throughout this dissertation.

I argued at the beginning of this dissertation that too often the entire meaning of software is represented by its functional meaning (i.e., the series of ones and zeros that signify software code). Limiting the meaning of software in such a way fails to account for the ways in which the discourse of the architects of code within the software programming culture encode software ideologically, a process that I called the discursive coding of software. In recognizing the ways in which software is discursively coded, I suggested that lay users of software can cultivate a digital literacy of software. Such a critical literacy of the digital offers avenues by which lay users can become “questioners of technology” (Selber, p. 121). Becoming questioners of technology stands to transform lay users from passive subjects, unaware of, or interested in, the discursive coding of the software they use on a day-to-day basis to reflective users, and possibly even evangelists, who recognize the ways in which ideology functions through code and users of software.

Before I offer examples of what digital literacy on the part of lay users might look like in action, I first want to revisit the answers that I constructed as a result of my analyses for the following two questions: 1. What are the political ideologies encoded in software? 2. What are the practices by which these ideologies function in the software programming culture?

Using the spatio-temporal framework as a guide in my choice of methods and texts to analyze, I began in Chapter Three to uncover practices by which the discursive coding of software by the software programming culture occurs. Specifically, I identified and described the shared cultural practices that constitute this culture and provide the means by which to create meaning(s) for software. Through an emphasis on iteration (Emirbayer & Mische) and the social positionings organized around this temporal orientation to the past, I concluded
that this network of practices, which includes the practice of programming, instrumentalist talk about coding issues, evangelism, and practices for regulating software use, provide stability to the culture and its members. I also concluded that this network of practices serve to differentiate the software programming culture from other cultures and gives its members the tools by which ideology functions.

In examining how these practices are routinely “reactivated” (Emirbayer and Mische) by actors, I identified two kinds of software technology: established software technology and disruptive software technology. Members of the software programming culture create established software technology by reactivating the network of social practices in a particular way, through a linear development process, the creation of transparent source-code software, diffused evangelism, and copyright. Disruptive software technology, on the other hand, is created through the reactivation of this network of practices; these reactivations typically include linear development, black-boxed source-code software, structured evangelism and regulatory practices, including trade secrets, patents, and copyright. Analyzing the ideological significance of these different kinds of software created through the relatively unreflective practical activities of members of the software programming culture was the aim of the next two chapters.

In Chapters Four and Five, I analyzed the political ideologies encoded in software. By examining the ways in which the discourse of spokespersons from different communities within the software programming culture construct trajectories of action meant to achieve in the future certain ideological values, I illustrated how spokespersons, even before they were spokespersons, imagined through their projectivity ways in which the practices of the software programming culture could facilitate this ideological achievement. Gates, Stallman, and Torvalds, though members of the same culture, encode the shared cultural practices and the object of software very differently—Gates according to the ideology of capitalism, Stallman according to the ideology of idealism, and Torvalds according to the ideology of pragmatism.

As Chapter Five illustrates, these always competing, though not always antithetical, ideological trajectories are reified by collectives of actors that form different communities organized around these ideologies: the proprietary software industry, the Free Software
community, and the Open Source community. The reproduction of the discursive coding of software by these communities pointed to the ways in which members of the same culture align themselves with and against certain ideologies according to the respective identity these actors internalize (Castells). Consequently, these identities, which I described in a modified form of Castells’ identity taxonomy, included the legitimizing identity of the proprietary software industry; the resistant/would-be-project identity of the Free Software community; and the resistant/would-be-legitimizing identity of the Open Source community. Each of these communities arose out of the coalescing of actors with a particular shared identity that aligned them with one another while, at the same time, aligning them against actors who shared a different identity. Through the roles as programmers and the practices available to them as members of the software programming culture, different communities arose out that sought to stabilize certain meanings of software and to change other meanings.

In Chapter Six, I analyzed the deliberative rhetorical action that I identified as resulting from actors’ “practical evaluation” (Emirbayer & Mische), which is mediated by iteration and projectivity and the social positionings created by each. This analysis revealed the ways in which actors’ can reflectively “reconfigure” (Emirbayer & Mische) the structured network of social practices in ways that allow actors to enact their ideological beliefs. Analyzing the different applications of the genre of software licensing agreements associated with each of the different communities of the software programming culture, I located the construction and continual use of the Microsoft EULA and other proprietary software licensing agreements by the proprietary software industry, the Free Software Foundation’s GNU GPL by the Free Software community, and the Open Source Initiative’s meta-license the OSD by the Open Source community, as examples of strategic and deliberative action on the part of the communities in each of their respective efforts to further stabilize their own ideological meaning encoded at the expense of the efforts by the other communities. The result of these deliberative actions, in which the structured regulatory practice of copyright was reconfigured, is the cementing in law the ideological trajectories evangelized by these communities, as well as the sets of relations between these communities.

By adhering to the guidelines provided by the spatio-temporal framework, I was able to map a network of meaning for software and the software programming culture. Attention
to the ways in which space and time shape and are shaped by discourse allowed me to uncover the ways in which cultural practices both stabilize the software programming culture and offer the means by which to create significant ideological difference, a “plurality of codes” as Hall states (Grossberg, 1996a, p. 137), within that culture. In short, the spatio-temporal framework facilitated an analysis that emphasized the interplay of structure and agency and the ways in which that interplay creates and recreates a context for social action, a context that both constrains and empowers and actions of actors. It is this interplay, after all, that theories of practice seek to uncover, regardless of critiques that challenge these theories’ ability to do so.

So how does knowing the processes by which software is discursively coded by the software programming culture facilitate the identification of means by which lay users can develop digital literacy? My analysis of the discourse of the software programming culture reveals the competing ideologies encoded in software and the practices by which these ideologies function. Knowing what these ideologies are (e.g., the ideology of capitalism, the ideology of idealism, and the ideology of pragmatism), and understanding the relationship of one ideology to another, can allow lay users to think about the ways in which the ideology discursively encoded in the software that they use is reflective or is not reflective of their own identity and the ideology according to which they live their own lives. As questioners of technology lay user might ask themselves: Does the use of established software technology encoded according to the ideology of capitalism, such as Microsoft Windows reify my own values? If there is incongruence between the ideology encoded in Windows and my own ideology, what might I do so that my software use is aligned with your values? Would using the disruptive software technology OpenOffice, which is an open-source project copyrighted by an OSI-certified licensing agreement, achieve this alignment because of OpenOffice’s pragmatic encoding? Will I call the free and open-source operating system “GNU/Linux” or “Linux,” and what does my choice mean for me in terms of my own ideological construction? Asking questions such as these constructs lays users as “questioners of technology,” (Selber, p. 121), a first step in digital literacy.

Becoming questioners of technology, generally, and questioners of the discursive coding of software, specifically, opens up the possibility of lay users participating in the
discursive coding of software. To do so, I do not naively suggest that lay users go out and become software programmers. This is not a realistic prospect. As my discussion of the relationship between the practice of programming and nerd/geek technophilia, most programmers are called to this endeavor, in the same that others might be called to teach or to sing. Though certain practices already sanctioned by their everyday use in the software programming culture will never provide feasible means by which lay users can participate in the discursive coding of software, some practices do; namely, the practice of evangelism and the practice of copyright.

As illustrated in the evangelism of Gates, Stallman, and Torvalds and the communities for whom each is a spokesperson, engaging in discursive practices with the aim of evangelizing the ideological encoding of software is a practice that lay users of software can themselves practice. As a lay user of software, my own research is an act of evangelization in that it seeks to unveil the negotiations of power that occur through the discourse of the software programming culture. Though the opportunity to spend six years researching these negotiations is not a privilege that all, or even most, lay users can take advantage of, reading the accessible texts that circulate among members of the software programming culture and talking about them gives lay persons an entry point to evangelize about the ideological significance of software. Slashdot (http://www.slashdot.com), where “news for nerds” is posted and discussed, includes a great number of accessible texts that report and comment on the activities of those in the hardware and software programming culture, as well as the effect of these technologies in society. This site also gives everyone, experts and lay users alike, the opportunity to report and comment on these technologies. Though a tough skin is definitely needed for a foray into the conversations on the site, especially if the topic of conversation is a hotly contested one, Slashdot and other sites like it give lay users the knowledge and opportunity to evangelize.

And just as the communities of the software programming culture support and reify through their use particular software licensing agreements that correspond to the identity and ideologies that each community aligns itself with, so to can lay users of software. Reflectively choosing which software licensing agreements you will and will not agree to is an act of digital literacy. Of course, making such decisions demands actually reading the
agreement, something that I am willing to bet most lay users of software do not do. After all, in the heat of wanting or needing to use a piece of software for the first time, the demand by the program’s licensor that the software licensing agreement be accepted before the program can actually be used is typically an inconvenient annoyance. However, as I have sought to illustrate in this dissertation, those licensing agreements are significant and construct licensees’ software use, and hence the licensees themselves, ideologically.

Another means by which lay users can participate in the discursive coding of software is through the supporting roles that some lay users provide to the software programming culture. Specifically, I am thinking here of professional communicators in both the workplace and academia. Writing software documentation, creating manuals, and conducting usability testing of software are some of the important services that professional communicators undertake in the process of producing a successful experience for the lay end-user of software. And educating professional communicators in the practice of performing these tasks effectively is the responsibility of many professional communication instructors. Consequently, as a professional communication instructor myself, I argue that when we offer examples or create assignments having to do with software use, we must not only talk about the possible writing strategies for creating effective documentation but also talk about the ways in which this documentation supports software that is ideologically encoded, a process that ideologically encodes the documentation as well.

We need to offer examples and encourage students to offer examples of the ways in which software documentation is encoded. For example, end-user documentation for most proprietary software, including Microsoft software, is also proprietary. However, documentation for free and open-source software is often shareable in the same ways that the software is. For instance, the Free Software Foundation constructed and evangelizes the GNU Free Documentation License. Software licensed under this agreement is copyable, modifiable, and redistributable. And the documentation and manuals for the software of open-source software distribution and service corporation Red Hat is sometimes licensed under the Open Publication License (OPL), which is more restrictive than the GNU Free Documentation License but not as restrictive as proprietary documentation licensing agreements. For example, the OPL stipulates that licensees must receive permission for the
Further research on the discourse of the software programming culture also stands to extend the digital literacy of academics and their audiences. Research that re-examines the continued use of the shared cultural practices that I specified as pivotal the software programming culture would potentially serve to complicate the analysis I conducted in this dissertation by approaching these practices from a different theoretical perspective or analytic framework. And additional research that analyzes other cultural practices by which the discursive coding of software occurs within the software programming stands to complicate this network and its role in providing structured continuity to the culture, while, at the same time, that is provides resources by which actors to reconfigure can bring their identities to bear on the software programming culture. As my discussion of the role of professional communicators in creating documentation and manuals for software illustrates, research into the relationship between software and its supporting texts stands to further develop digital literacy. By encouraging students, workplace professionals, instructors, and academic researchers to understand how the software-supporting documents they create are also ideologically encoded in ways that reify the a particular ideological trajectory and not others, the development of digital literacy can become just as important as the development of effective writing strategies.

Through the examples offered here, I seek to confirm that lay users of software do, in fact, have the means not only to develop a digital literacy of software by becoming questioners of software technology but also to act on that digital literacy through some of the very same practices available to members of the software programming culture; namely, evangelism and regulatory practices. Additionally, lay users have public arenas in which they can participate in the discursive coding of software not only with other lay users but with members of the software programming culture. Further, certain lay users, such as professional communicators who provide support services and products to the software
programming culture, have a unique opportunity to participate in the discursive coding of software through the creation of workplace writing aligned with or against ideologies encoded in software.

We have a unique opportunity to expand research into the very beginnings and early consequences of monumental technological shifts that are changing the world, to open the space of critical inquiry, and, in doing so, to broaden participation by all in the ways these technologies are changing the social, cultural, economic, and political landscape. My contribution to this effort attempts to make visible the relatively invisible culture of software programming by drawing attention to the ways in which these architects have discursively coded not just software but our very beings.
REFERENCES


Microsoft. (1998). End-user license agreement. (Systemroot\System32\eula.txt.)


