ABSTRACT

BROCK, KEVIN MICHAEL. Engaging the Action-Oriented Nature of Computation: Towards a Rhetorical Code Studies. (Under the direction of David Rieder and Susan Miller-Cochran).

I argue that, at the convergence of rhetoric, software studies, and critical code studies, there is a space for an approach to scholarly inquiry best described as "rhetorical code studies." While there have been efforts by critics in each of the aforementioned fields to approach the questions asked by the others, there has of yet been no explicit attempt to articulate how rhetoric's interest in persuasive action could be located in software and code as objects of study, nor has there been an effort to express how that study could be explicitly dealt with in rhetoric. In Chapter 1, I locate rhetorical code studies in the midst of recent engagements with cultural investigations of code and with rhetorical inquiry into digitally-mediated forms of communication and I outline how rhetorical code studies would be valuable to rhetoricians, software scholars, and critics of code alike.

Drawing connections between rhetoric and algorithmic computation back to classical Greece, in Chapter 2 I position the *enthymeme* as a rhetorically-oriented algorithm computed by a given audience, and I demonstrate how contemporary software developers use enthymematic reasoning to persuade their colleagues to engage in particular types of development practices. As discussed in Chapter 3, there are certainly conventional forms of discourse surrounding the composition of code texts wherein developers attempt to influence one another about their means of development—using methods that only rarely focus on *logos* as the primary appeal, with most developers relying instead on a mix of *ethos* and *pathos* to make their cases. However, the code texts *themselves* are forms of persuasive arguments, and Chapter 4 focuses on the code related to the Mozilla Firefox web browser as

a set of rhetorical texts that operate in some ways that are similar to and distinct from more conventional forms of communication. The strategies used by developers in code often rely on more heavily implied enthymemes than in natural language discourse, but they offer no less intriguing ways of understanding how computational logic can be used to persuade both developer (in terms of continuing to build a given program) and user (in terms of how those computational activities will be executed in the compiled program).

Ultimately, I offer several potential trajectories for the nascent field of rhetorical code studies, connecting those possibilities to current trends in scholarship on procedural rhetoric, genre studies, and activity theory. Given the range of perspectives already available for inquiry in rhetorical code studies via rhetoric, software studies, and critical code studies, these proposed trajectories can only benefit our understanding of the numerous, varied, and *significant* roles played by code in contemporary communicative activities.

Engaging the Action-Oriented Nature of Computation: Towards a Rhetorical Code Studies

by
Kevin Michael Brock

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Communication, Rhetoric, and Digital Media

Raleigh, North Carolina

2013

APPROVED BY:

David Rieder Co-chair of Advisory Committee	Susan Miller-Cochran Co-chair of Advisory Committee
Jason Swarts	Kenneth Zagacki

DEDICATION

To my best friend and partner, Erin Anderson. I love you. Thank you for always being there for me.

BIOGRAPHY

Kevin Brock was born in Mansfield, Ohio in 1981. He graduated with a Bachelor of Arts in English from East Carolina University in 2003 and pursued graduate studies at North Carolina State University, achieving a Master of Arts in English in 2007. During that time, his research interests turned to rhetoric and composition, with a focus on the role played by digital technology on composing practices. After several years as a lecturer in the Department of English at North Carolina State University, he enrolled in the institution's Communication, Rhetoric, and Digital Media program, where he studied software code languages as forms of rhetorical communication. He has published articles and reviews in journals such as *Enculturation*, *Technoculture*, and *Kairos* and presented at major conferences including the Conference on College Composition and Communication, the annual convention of the National Communication Association, and the Computers and Writing conference. With David Rieder, he has displayed interactive digital media projects at the Contemporary Art Museum in Raleigh, NC; Duke University in Durham, NC; and the 2012 Conference on College Composition and Communication held in St. Louis, MO.

ACKNOWLEDGMENTS

The number of people that I could, and should, acknowledge here for their support and friendship is far higher and would take up more room than I have available. I apologize for not being able to thank everyone individually who has helped and guided me to, and through, this project.

Let me begin by thanking the members of my committee, whose input and guidance have been invaluable. To Jason Swarts and Ken Zagacki, I offer my gratitude for their patient feedback as well as their advice on preparing to join the profession. Susan Miller-Cochran has been a mentor and friend for six years, and I dearly cherish that relationship. Her counsel as a scholar and an administrator has been extremely helpful to me in seeking out new directions for research and balancing my various duties in the process. I am proud to call David Rieder a friend as well as a collaborator, and I have learned much from my time as his student and colleague that I suspect I would not have learned elsewhere. I am indebted to his willingness to help me work through numerous projects that tested my capabilities as an emerging scholar.

I want to acknowledge the role that Nancy Penrose played in the development of my career; as the individual who trained me to be a teacher, I cannot overstate the influence she has had on my pedagogy—to say nothing of my interest in rhetoric as a discipline.

My thanks are also offered to the many colleagues I have worked with as a master's and doctoral student and whom I have come to consider as close friends. Dawn Shepherd, Kathy Oswald, and Kati Ahern deserve special note as peer mentors whose approaches to research and teaching have influenced my own, while Matthew Davis and Casey Boyle have served in exceptional capacities as both friends and exemplars.

In addition, I want to recognize the Communication, Rhetoric, and Digital Media students currently pushing the bounds of research in rhetoric and digital media in novel and exciting ways. I cannot wait to see the directions in which Ashley Kelly, Kate Maddalena, and Fernanda Duarte take their emerging careers, and I am thankful for the conversations that I have shared with each. I look forward to sharing many more with them in the future, and I hope that the CRDM program continues its tradition of developing and training the finest junior scholars studying communication and rhetoric.

Finally, I want to celebrate the impact that my family has had on my work and life. My parents, Jim and Cindy, have both been tremendously supportive of me in everything I've done in life, and I love and thank them dearly for it. My brothers David and Patrick have always been on hand to offer relief and sympathy when needed. Erin, my partner, has been my best, closest, and dearest friend as well as my resolve whenever I have been unsure of myself. I cannot thank her enough for the love she has given me so freely and fully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION TO RHETORICAL CODE STUDIES

At the intersection of the disciplines of digital rhetoric, software studies, and critical code studies, there exists a blind spot in which focused critical inquiry could and should be developed. This blind spot is defined by a lack of focused engagement with one of two approaches to the study of digital software, the code that it comprises, and the procedural logic that powers computer technologies. The first approach relates to rhetoric, and the second involves software and critical code studies. Regarding the first, rhetoricians have tended to ignore or eschew the computational qualities of digital media, preferring instead to focus on the end-user interfaces with which the technology-using populace most regularly interacts. Regarding the second, critics of software and code have generally overlooked the rhetorical qualities of the software programs and the logical processes that make up code; instead, they primarily apply cultural and literary lenses of inquiry to their objects of study. The blind spot at the convergence of these fields can and should be explored so that it can serve as a focal point for critical scholarship on the rhetoric of computation and the code that facilitations computational activity.

Concerning the lack of focused engagement with computation by rhetoricians, current practices related to the rhetorical study of digital media center on increasingly ubiquitous software applications. There are several approaches to multimodal and multimedia text production and criticism to which rhetorical scholars contribute. First, there is a considerable body of scholarship dedicated to the combination of image and text, especially in regards to the primarily visual nature of web-based documents (Fleckenstein, 2003; Kress & van Leeuwen, 2006; Sorapure, 2010). Kress & van Leeuwen (2006) emphasized rhetorical arrangement in particular as a significant mode of communication, focusing their efforts on rhetorical decisions related to visual arrangement and document design. A second group of scholars, of whom Shipka (2005; 2011) and Wysocki (2005) are among the most prominent members, has explored the boundaries of multimodality through physical as well as design-based components of texts. For example, Wysocki (2005) examined the implicit constraints always exerted upon rhetors, extrapolating her conclusions from an example situation in which crayons might be considered acceptable writing tools for students' academic coursework. Yet another initiative focuses on multimodality as it relates to particular rhetorical situations and rhetors' needs; Yancey (2004) observed that regular, day-to-day writing involves far more modes of communication and technologies than are generally addressed in writing courses and research and called for an embrace of broader definitions of "writing."

This set of approaches is limited because attention has inadequately been paid to the myriad influences of particular software tools on rhetors' inventive processes. Until very recently, the closest that many scholars (including Bruce & Hogan, 1997; Blythe, 2001; Fleckenstein, 2005; Payne, 2005) had come to addressing these concerns was by offloading critical inquiry to Selfe & Selfe's 1994 examination of the sociopolitics of technological interfaces. Selfe & Selfe observed that interfaces in electronic environments communicated particular social, cultural, and political values that reflected developers' assumptions.

Through an examination of these assumptions, it became possible get a sense of those whom developers perceived as the expected user populations of those interfaces and what sorts of knowledge, technical proficiency, or other forms of awareness the users were expected to possess (or that were implicitly demanded of them) when using those interfaces (Selfe & Selfe, 1994). Grabill (2003) articulated the difficulty possessed by most rhetoricians in regards to undertaking the work of identifying rhetorical activity in and through interfaces, pointing out the distinction between interface and conventional writing or speaking as a site of rhetorical communication:

> [I]nterfaces are difficult to talk about. They seem natural and inevitable to most people. They are often transparent. Students in my classes can't imagine computers being any other way—and most of the time, neither can I. Interfaces are what programmers write. (pp. 465-466)

In essence, rhetoricians have often avoided overtly technological concerns simply because those concerns are perceived to be *outside the bounds* of their disciplinary study. For Grabill (and, implicitly, for many others), interface construction, and programming in general, are clearly forms of "writing"—and thus a form of making meaning rhetorically—but they are not the sort of writing that could be *easily* addressed in a course on rhetoric.

Despite this general difficulty in incorporating technological interfaces into rhetorical study, in the past several years a small number of rhetoricians (among them Carpenter, 2009; Arola, 2010) has attempted to address the cultural and ideological value systems inscribed into software interfaces—to say nothing of their code—by the developers that created those programs. There is one notable exception to this group. Haefner (1999) made what might have been the first substantial attempt to move beyond Selfe & Selfe's work, when he argued that computer code possessed as significant a set of politics as the interface(s) engaged directly by users. As a result of this increasing set of technologically focused investigations, the broadly-defined population of contemporary rhetorical critics have made the argument that emerging technologies, as a general category (often simplified as “new media” or “digital media”), enable new forms of meaning-making, especially for citizen populations communicating via end-user interfaces. However, this same population often overlooks that specific technologies *themselves* (e.g., individual operating systems, word processors, or programming languages) are similarly nuanced components of meaningful construction within the fundamental category of technological mediation.

Meaning, in this set of contexts, is geared towards *developers* of those technologies and interfaces rather than the broader set of users whose activity is constrained by developers' actions. If rhetoric is, as noted by Aristotle, the “ability […] to see the available means of persuasion” (2007, I.ii.1), then rhetoricians would benefit by recognizing the limitations in their work, the possibilities of which are being explored by scholars in software studies and critical code studies. A rhetorical code studies can potentially incorporate within its fold multiple types of rhetorical activity and interaction, including developer-to-developer persuasion, technology-to-developer persuasion, technology-to-user persuasion, and developer-to-user persuasion. Much of current studies related to digital rhetoric focus on some components of the last two categories in this list; for the purposes of this project, and explained in more detail in Chapters 2 and 3, I emphasize the rhetoric of developer-to-developer persuasion since the rhetorical activity in that sort of interaction has a significant impact on subsequently-triggered sites of activity (such as technology-to-user forms of persuasion).

The second relevant approach to the study of software for a rhetorical code studies represents a lack of focused engagement with the rhetorical qualities of software and code by scholars associated with the fields of software and critical code studies. Despite a relatively superficial acknowledgment of the role of rhetoric in software development and the use of computer technologies, software and code critics generally fall into one of two camps. The members of the first seem unaware of the acknowledgment(s) of rhetoric they make in relation to their objects of study. The members of the second seem uninterested in exploring the connections between rhetoric and software even as they discuss, sometimes explicitly, how *meaning* is constructed for developers and other practitioners of software in and through code—especially given the domain-specific context of meaning in and through the activity of software development. A number of scholars studying software and code have implicitly noted interest in rhetorically significant qualities of software and computer technologies, but none of these researchers has explored those rhetorical qualities to any extent. Wardrip-Fruin (2009), for example, suggested that computer logic has rhetorical value in his description of digital media processes as "humanly meaningful" (p. 156). Wardrip-Fruin illustrated this argument by observing how the procedural logic powering the generative fiction program *Tale-Spin* were hinted at through the structures of the stories it produced (p. 130). Kitchin &

Dodge (2011) identified multiple levels of software activity that held significance for networked systems of humans and technologies alike: coded objects, coded infrastructures, coded processes, and coded assemblages. Each of these levels, they argued, affect contemporary life in interrelated ways. For example, coded objects, which necessitate software in order to function (from cell phones to DVDs) facilitate particular activities to which we attribute meaning, and these activities are linked together by a number of coded infrastructures (which partially or fully make use of software to run), including transportation, financial transactions, and the use of sewer systems (Kitchin & Dodge, 2011, pp. 5-7). In an analysis of the Microsoft Word mascot and help utility “Clippy,” Fuller (2003) focused much of his critique on the interface-oriented appeals made by Microsoft to its users about how to use the program properly. For Fuller, “Clippy” epitomized a failure on Microsoft's part to recognize the nuanced, contextual activities for which users would write or otherwise use Word. Instead, Microsoft exerted its own sense of proper use through the Word interface and through the “Clippy” tool, which guessed at the intent of—and implicitly judged—users and their efforts.

While the focus thus far has been centered on the individual failings of rhetoric, software studies, and critical code studies, there are two scholars whose work *has* addressed —albeit implicitly—the goals of, and thus set the stage for the emergence of, rhetorical code studies as a convergence of all three of those fields. The first scholar to approach a rhetorical code studies is Ramsay (2011), whose “algorithmic criticism” will be addressed later in this chapter and more fully in the next chapter. Ramsay's approach to criticism, briefly described, explores the possibilities of invention facilitated by computational procedures for both creative and critical practice. These possibilities for are especially exciting in that they can lead scholars to generate entirely new texts and forms of communication as well as reconceptualizing existing texts.

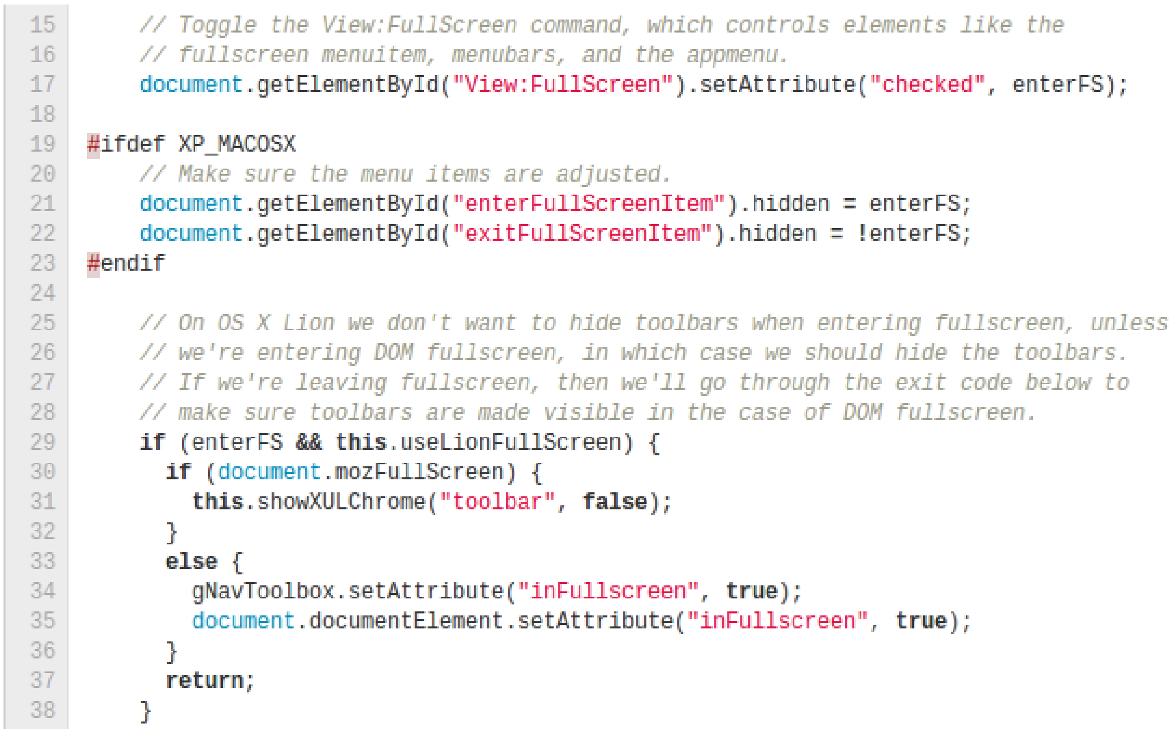
The second scholar to be noted here, and the critic who has come closest to illuminating this point of convergence between rhetoric and the study of software, is Bogost (2007), whose concept of *procedural rhetoric* has articulated a means of persuasive influence demonstrated through particular media—such as video games—as a way of teaching audiences how to *use* those media. Procedural rhetoric, according to Bogost (2007), is "the practice of using processes persuasively" (p. 28). This definition is most commonly considered in regards to the expressive outcomes of processes (i.e., software interfaces), but procedural construction is itself persuasive as well, setting up *how* processes can potentially be used by skilled rhetors. The procedural workflow of accomplishing certain tasks is and can be used as a persuasive medium, and a rhetor can construct a complex argument *in code* which unfolds through the expression of relevant processes. Bogost noted that, for procedure, "arguments are made not through the construction of words or images, but through the authorship of rules of behavior, the construction of dynamic models. In computation, those rules are authored in code, through the practice of programming" (p. 29). By focusing on the potentiality of dynamic computation—when an audience attempts to explore the procedures composed (i.e., programmed) by a rhetor for a digitally-mediated situation—the means for, and types of, action to be influenced by suasion are drastically altered by user and computer system alike.

In summary, over the past decade there have been significant critical forays into the cultural and critical implications of software and code as well as into the rhetorical qualities of digital media. However, there has been little scholarship yet that explicitly bridges the gap between these types of approach. The proposed means of connecting together the study of rhetoric and software is through a *rhetorical code studies*, in which computer software and the code that makes it up are understood as rhetorically powerful forms of making and communicating meaning.

1.1: Rhetorical Code Studies

Rhetorical code studies combines multiple aspects of the study of rhetoric (in particular, rhetoric as meaningful communication) and of software and whose focus of inquiry centers on the following two characteristics of code and its discursive contexts. The first is the rhetorical qualities and capacities of software code, which include both source code and executable programs. Source code is the readable set of operational commands written in any number of existing computer languages. Today, most source code is written in a "high-level" language, meaning that it is immediately more accessible to humans than to machines: high-level source code must be compiled (that is, translated) into a lower-level language in order to be executed by a given machine. High-level languages, such as C, Java, and Ruby, possess syntax and vocabulary roughly approaching that of a natural language like

English, while low-level languages like assembly languages and machine code specify far less human-readable instructions to the machine(s) executing their commands. The second characteristic to be addressed is the discourse surrounding the development and use of code, which includes code-level comments and meta-commentary. Developers regularly communicate with one another inside lines of code in statements called comments, or non-interpretable, non-executing natural language statements. For example, see Figure 1.1, below; in this excerpt from code for the Mozilla Firefox browser that renders the program when in fullscreen mode, lines beginning with // are comments from one developer to another. Specifically, the commenter has articulated what his or her goals were in constructing the function that follows the block of comment lines.



```
    // Toggle the View:FullScreen command, which controls elements like the
    // fullscreen menuitem, menubars, and the appmenu.
    document.getElementById("View:FullScreen").setAttribute("checked", enterFS);

#ifdef XP_MACOSX
    // Make sure the menu items are adjusted.
    document.getElementById("enterFullScreenItem").hidden = enterFS;
    document.getElementById("exitFullScreenItem").hidden = !enterFS;
#endif

    // On OS X Lion we don't want to hide toolbars when entering fullscreen, unless
    // we're entering DOM fullscreen, in which case we should hide the toolbars.
    // If we're leaving fullscreen, then we'll go through the exit code below to
    // make sure toolbars are made visible in the case of DOM fullscreen.
    if (enterFS && this.useLionFullScreen) {
      if (document.mozFullScreen) {
        this.showXULChrome("toolbar", false);
      }
      else {
        gNavToolbox.setAttribute("inFullscreen", true);
        document.documentElement.setAttribute("inFullscreen", true);
      }
      return;
    }
```

Figure 1.1: Comments within code for Mozilla Firefox. Comments are on the grayed-out lines beginning with //.

The comments exist among, but “outside,” the lines that comprise the compilable software program. In addition to intra-code commentary, developers actively engage each other just as often in more conventional avenues of discourse outside the code, such as via email lists, bulletin boards, and the like. As will be demonstrated in Chapter 3, code files can serve as fascinating and significant sites of rhetorical action (primarily focused on practices of software development) both in and “between” code operations.

It is important to engage the possibilities for rhetorical code studies at the present because both of these types of discourse are integral to the construction and development of software. However, the significance of code and code-related discourse, as meaningful forms of communication in addition to influential agents of culture and sociopolitics, has not been appropriately acknowledged and examined by the groups that should be most critically interested and invested in *how* code and code-related discourse operate. In addition to providing an avenue for focused critical inquiry of unexplored objects of study, rhetorical code studies would offer new means of engaging in the creative and rhetorically informed *production* of meaningful code and software. This is not to suggest that existing software is not already meaningful, but instead I posit that software whose code has been composed with as much rhetorical awareness as its intended execution and output could herald novel and valuable ways of approaching computer use in general and code composition in particular.

This project seeks to shine a light on this point of convergence so that the emerging field of rhetorical code studies might be more fully scrutinized, as software and code can help us understand more accurately how we attempt to communicate meaningfully with one another across digital contexts. For rhetoric in particular, a field in which scholars seek out knowledge of and proficiency with the available means of persuasion for a given situation, it is alarming that there exists a general lack of interest in the broader possibilities of meaning-making in and through digital technologies. Further, given the increasingly significant role that software plays in the daily activities of all manner of individuals and populations, it is imperative that critics of rhetoric and software alike understand how software and its code exert influence upon our efforts to communicate with one another so that we—academics, software developers, and the general public—can make more effective and aware use of these code-based forms of meaning-making.

In the chapters that follow, this potential field of rhetorical code studies will be developed through a focus on a set of interrelated concepts emphasizing the rhetorical means and goals of code. The first of these is *action* as understood through Burke's (1969) term *symbolic action*. For Burke, symbolic action refers to any communicative effort whose meaning extends beyond the specific set of acts that makes up the transmission of a particular message. The second concept central to my argument is *persuasion*, the attempt by one or more rhetors to influence an audience to act in response to a particular message. The third and final major concept to be discussed here is *agency*, the quality of both rhetor and audience to engage themselves in a rhetorical act; Miller (2007) referred to the quality of agency as "the *kinetic energy* of rhetorical performance" (p. 147). Ultimately, an examination of these concepts will serve as the basis for a critical analysis of the Mozilla Firefox web browser, whose long-term, open source development provides an accessible point for a rhetorical inquiry into its code.

From the initial point of convergence identified above, rhetoricians and scholars of code alike can move forward along a number of trajectories that would provide insight into the meaningful, rhetorical, and critical values of computational objects of study. In order to demonstrate how a rhetorical code studies is derived from the convergent disciplines of rhetoric, software, and critical code studies, the remainder of this chapter continues with an in-depth description of these two fields and the scholarly conversations taking place in them.

1.2: Software Studies

The field of software studies is focused primarily on the social and cultural significances of, and influences upon, the processes of software programs and the logic that facilitates their use. There are several major initiatives that are currently being developed by software scholars beyond the set of varied approaches to the study of software by critics who affiliate themselves with the general discipline. The first of these is cultural analytics, which focuses on studying how software can be used to explore the cultural significance of massive bodies of data. The second is expressive processing, an approach to software studies interested primarily in the logic of software as it is expressed for particular purposes, from video games to electronic literature to corporate accounting. In addition, platform studies is a field closely related to software studies but emphasizes the critical value of systems of software and hardware as ecological objects that illuminate how decisions related to software programs are constrained by these interrelated components.

On a large scale, software as an object of inquiry involves global networks like Google's search mechanisms and cloud-based information storage or cell phone network infrastructures; on a smaller but no less significant scale, software studies is concerned with computer functionality like that of the *loop*, in which elements of a data set are iteratively manipulated by a set of operations for a particular set of purposes. Manovich (2008b) defined the goal of software studies as “investigat[ing] both the role of software in forming contemporary culture, and cultural, social, and conomic forces that are shaping development of software itself” (p. 5). In the early 2000s, Manovich (2001) and Fuller (2003) called for the creation of a field whose critical study was the developing body of computer technologies and the software programs they ran. Fuller led the first workshop, in 2004, toward this end, and it resulted in the field's first edited collection on the subject and the first publication making use of the term “software studies.” The collection was separated into micro-essays on specific concepts related to software, such as elegance, function, loop, and variable. Manovich organized a follow-up conference in 2008; this second workshop provided a forum for many current points of interest in software studies to establish themselves across a number of disciplines, from art (Crandall, 2008) and design (Lunenfeld, 2008; Sack, 2008) to science and technology studies (Bowker, 2008) and literary studies (Douglass, 2008). In addition, several scholars presented the initial grounds for the emerging software studies sub-fields of expressive processing (Wardrip-Fruin, 2008) and cultural analytics (Manovich, 2008a). Marino (2008) was present to discuss the still-nascent field of critical code studies (about which more is discussed later in the chapter). Further, Bogost (2008) introduced platform studies as a means of studying video games as ecologies of hardware and software systems.

Platform studies deserves special note as a field closely related to software studies due to its focus on the ecologies of software and hardware technologies which serve as the basis for software activity. For example, where a software scholar might be interested in the cultural values enabled by a particular programming language, a platform scholar would focus on how a given hardware system (like a desktop computer with a 32-bit processor running the Windows XP operating system) constrains the sort of software texts, or set of processes, that could be created or disseminated through that hardware system. Unlike software studies' emergence from a general call for the study of digital media, platform studies was formed as a means for video game scholars to draw attention to the technologies that enabled the play of specific games. The most notable studies in this regard include Bogost (2008), Montfort & Bogost (2009), and Sample (2011); each has pushed the traditional bounds of game studies as interactive forms of narrative in order to highlight the roles of game *systems* in the development of games as meaningful texts. Platform studies demonstrates the potential for rhetorical code studies in its goal of critical investigation into the relationship between game creation, play, and technological infrastructure. However, where platform studies is dedicated to the application of its criticism towards video games, rhetorical code studies holds within its scope the universe of software and code that software studies attempts to address.

Most scholars who associate their goals with those of software studies do so in a relatively unrestricted fashion, noting interest in some particular political, social, or other cultural study of software at one or more levels of technology, e.g., the graphical user interface, high-level programming languages, or even the low-level assembly languages that translate readable code into executable operations to be run by a machine. For example, Parikka (2008) examined the ability to *copy* through digital software as both a new means of high-fidelity reproduction (as a command and as a tool embedded in various software programs) and as cultural technique that follows a tradition of quotation and recycling for the purposes of disseminating information. Many of the restrictions that are exerted upon software scholarship are built upon the qualities of new media outlined by Manovich (2001), which he argued were requisite for any scholarly understanding of how digital technologies worked:

1. new media objects are composed of digital code, which is the numerical (binary) representation of data;
2. the structural elements of new media objects are fundamentally modular;
3. automation is prevalent enough in new media systems that human presence or intervention is unnecessary;
4. new media are infinitely variable;
5. new media, as computer data, can be transcoded into a potentially infinite variety of formats. (Manovich, 2001, pp. 27-45)

Based on these principles, Manovich and other software scholars have explored the possibilities of digital technologies and data as means and forms of cultural construction and transmission. While not all software critics are interested in the technical qualities of digital media included in Manovich's list, these concerns nevertheless inform the vast majority of relevant scholarship, such as Cramer's (2005) analysis of algorithm as “magic,” in which he broke down the ways that various cultures have attempted to comprehend computation. Hayles' (2004; 2005) comparison of natural and code languages for meaningful purposes is also grounded in Manovich's principles of new media; Hayles suggested that code, despite its mutable, transcodable nature (which suggests a flexible or shifting potential meaning attached to it), does not possess the ability to signify or transmit multiple meanings in the liquid manner that natural language can and does. Fuller & Matos (2011), meanwhile, have extrapolated the possibilities of "feral" computing systems and the potential for wild, illogical designs that already emerge from the inherently logical nature of new media as data and code languages.

The general field of software studies includes within its fold a varied set of critiques, including Fuller's (2003) analysis of the infamous "Clippy" utility in Microsoft Word; Kittler's (2008) more general analysis of code as meaningful sub-systems of language; and Kitchin & Dodge's (2011) exploration of the multiple levels of activity of daily life into which software are incorporated, as objects, processes, infrastructures, and assemblages (p. 5). For many software scholars, there are several major components of a software program that indicate the systems of control and knowledge that its developers assume of, and impose upon, user bases, including the user interface, the language(s) in which the developers wrote the program, the systems in/on which the program runs, and even the potential uses for the software anticipated by the developers. By demonstrating the range of possible texts that could, and do, provide significant insight into how software and culture exert their reflexive influences upon one another, these software scholars have implicitly nudged the field as a whole towards the conventional domain of rhetoric.

1.2.1: Cultural analytics.

Turning away from the theoretical bent of general software studies is the more scientifically-oriented sub-field of cultural analytics, which makes use of the computational capabilities of computer technologies in order to map out culturally-insightful patterns from massive bodies of digital data. Manovich (2008b) made the initial push in this direction, arguing that the age of new media had been replaced by the age of “more” media, referring to the sheer amount of information that an individual could access easily with a computer. Cultural analytics shares a number of qualities with initiatives in the digital humanities, whose practitioners seek to develop tools for humanistic purposes. The primary difference between digital humanities and cultural analytics is that digital humanists focus on experimenting with the potential value that digital tools and software can bring to humanities-focused lines of inquiry. Moretti's (2005) visualizations of data from “distant reading” approaches to literary study, in which he looks at trends in literature across specific ranges of time, geography, and genre, fall under this mantle of digital humanities scholarship. In contrast, Manovich and the other cultural analysts are far less interested in the inherent humanities nature of the data being analyzed than the multiple ways that such data can be processed for potentially interesting ends. That is, cultural analytics emphasizes how digital tools can illuminate ways to identify and “read” value within a body of data; it does not articulate whether particular bodies of data are inherently valuable for cultural study. Instead, the data already exists, and it is only through analysis that interesting significance might be clearly determined.

For example, Manovich (2011) has experimented with “media visualization” as a way to scrutinize distinctions between elements of data sets, plotting out on a quadrant graph over 120 works each of Mondrian and Rothko to compare the mean brightness and saturation of each artist's oeuvre (see Figure 1.2, below). By transforming each painting into an (x,y) coordinate on the graph, Manovich was able to examine how each artist used particular hue sets and ranges of color within those sets, and he connected that data to broader cultural trends related to contemporary imitation of paintings (specifically, that since it was not expected by audiences the way it might have been several centuries prior, it was unlikely either artist would have a number of paintings that closely resembled one another). While this project has a clear object of humanistic interest, the artistic quality of the data is less valuable to Manovich than the sheer amount of numerate, transcodable data it offers: the paintings are transformed into sets of numbers which then are transformed into spatial points on a graph.

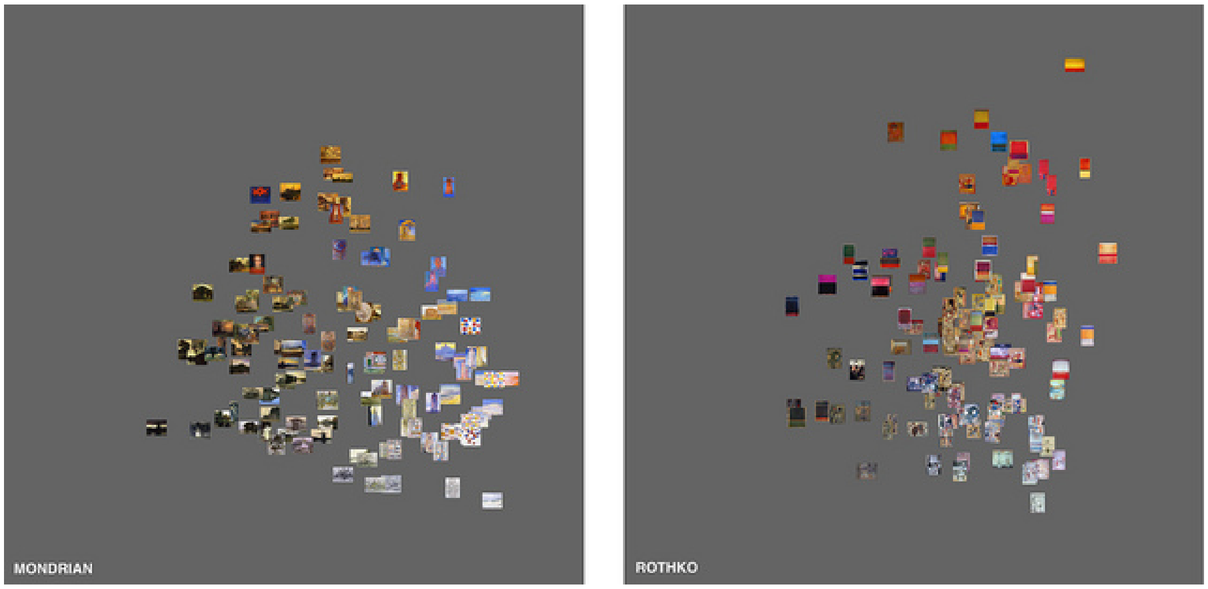


Figure 1.2: Graph comparisons of the works of Mondrian and Rothko performed by Manovich's cultural analytics lab. Paintings for each artist are clustered together based on mean brightness (x-axis) and mean saturation (y-axis).

1.2.2: Expressive processing.

Other scholars of software focus primarily on the types of processes whose logic fuels the use of software programs, such as the calculations that provide Google search results or the behaviors of computer-controlled video game characters. This approach is distinct from cultural analytics in that, for individuals studying the logic of computer processes, the ways of computing data, and what sort of meaningful results those computations generate, are of greater interest and value than what a particular set of data might offer for particular approaches to computation. For these critics, including Wardrip-Fruin (2009) who coined the term, the object of study is “expressive processing.” Because of its focus on how relevant technologies enable and constrain software and user behavior, his field has much in common with the related field of platform studies explored by critics such as Bogost & Montfort (2009). Scholars involved in this sub-field of software studies approach software as a way of "reading what processes express" and how processes "operate both on and in terms of humanly meaningful elements and structures" (Wardrip-Fruin, 2009, p. 156). For Wardrip-Fruin, video games and works of digital fiction provide an ample supply of texts (i.e., processes and narrative experiences) to explore, Bogost & Montfort (2009) explicitly specified that the discipline is not constrained to games as objects of inquiry, noting that even programming languages are types of platforms with underlying logic systems to be expressed through their use.

While there are some divergences between scholars in regards to the specific focal points of the field, there is one common line of agreement. No scholar involved in expressive processing has argued that technical knowledge of computer systems is *necessary* to perform successful inquiries into software processes, but almost all suggest that such a skill set is crucial to pursuing most fully questions of the sub-field. Wardrip-Fruin (2009) has described the problem as follows: "Trying to interpret a work of digital media by looking only at the output is like interpreting a model solar system by looking only at the planets" (p. 158). For many not familiar with how computers work, this statement may seem odd, but the point to take away is that a *solar* system has at its center a star rather than a planet, and it is the star that enables the entire set of planets to revolve around it and maintain their various ecological systems. For computers, programs (the planets in Wardrip-Fruin's analogy) require the framework provided by hardware and software alike in order for individuals to enjoy the interfaces they most often use.

1.3: Critical Code Studies

Where software studies employs the logic of computer processes as a means of studying both technology and culture, the more recently emergent field of critical code studies examines software *code* as meaningful text in its own right. Many critical code scholars are interested in exploring how particular code languages facilitate certain habits of mind as well as means of communication through the construction of code programs. There admittedly exists a significant overlap in research foci between software and critical code scholars; the latter sprung from the former as a directed effort to explore computer code as something more (i.e., more meaningful and significant) than its output, which early critical code scholars argued made up the majority of new media scholarship (Cayley, 2002; Marino, 2006). There are, as of yet, no distinct initiatives focusing on particular qualities or applications of code in the sense of cultural analytics and expressive processing as sub-fields of software studies. However, there do exist two broad camps of code critics: the first, epitomized by Marino, have focused thus far on reading code (and debating *how* to read code) as a type of literature. The other camp, led by Hayles, has primarily debated the potential distinctions between code, software, and natural language. A third direction of research prioritizes efforts to construct artistic texts called *codeworks* written in code and code-like languages.

Many critics of code view the field as an extension of literary criticism, with code texts providing insight into numerous conventional questions of scholarship, including authorial intent, contemporary trends in writing (programming) style and genre, or considerations of meaning within a particular body of code (Marino, 2006). Ramsay (2011) made use of code for such ends, performing an approach to criticism that enabled him to transform sets of literary texts into code-mediated expressions (referred to by literary critics as *paratexts*) so as to highlight novel means of interpreting those initial texts. For example, Ramsay compared the main characters of Woolf's *The Waves* by calculating the most frequently-used words in the vocabulary of each. By using these lists to shine light on the concepts most important to each character (correlating frequency with significance), Ramsay was able to outline a new act of critical reading. For Ramsay, algorithmic code serves as a novel means of engaging in scholarship by making familiar texts strange so as to generate new questions and avenues for critical research. Douglass (2011) has approached code somewhat more traditionally as a textual object of inquiry, questioning how code *is currently* read as a counter to studying how code *should be* read. That is, rather than suggest a "best practice" of code-related interpretation, Douglass has emphasized how we tend to evaluate and value code as a familiar or unfamiliar form of meaningful communication and what these existing practices mean for our ability to work with code in new and significant ways.

Other scholars, however, have begun approaching conventionally rhetorical, as well as literary, concerns, although they rarely explicitly associate their critical code work with rhetoric: specifically, critical code studies has prompted a trajectory into the potential for discovering what meaningful code texts can signify through their content, as well as through the processes they describe, for a variety of audiences. Burgess (2010) examined how the PHP script language changes the act of reading web pages and markup language, since PHP is interpreted by a web server and transformed into HTML before a user has the chance to see what it does when looking at a particular web page. Jerz (2007) explored both the source code for the late 1970s text-based video game “Adventure” and the physical location (Kentucky's Mammoth Cave) that inspired its creation. Jerz's analysis incorporated into its scope not just how the game attempted to replicate the author's experience of traveling through Mammoth Cave but how specific lines of code communicated significant meaning to the game player about how he or she should explore the possibilities for game play made possible by the author and his choice of code language. In addition, by visiting the physical site of Mammoth Cave, Jerz “played” by tracing out a path that someone in the video game might have virtually taken, observing the rhetorical failures in his physical performance in Kentucky to achieve particular types of action that, in “Adventure,” facilitated new situations and activities for the game player.

Yet another group of scholars straddles the boundary between software and critical code studies, emphasizing through their work how code and software are often as radically different objects of study as they are similar to one another. Chun (2011) has led this particular charge, pointing out that executing code—the compiled software that one runs in order to use specific programs on a computer—is distinct from its comparatively static source code that, while readable, does not act. For Chun, source code is an artifact that only hints at the possibility of what a program can do since the act of *using* the software cannot be replicated by *reading* its code. Hayles (2005) argued for a similar distinction, which she positioned between natural language and executable code language (which, for Hayles, comprised both compiled programs and their source code). Hayles' position focused on the ability of natural language communication to signify more than it literally suggests, while code merely described the computational operations that its compiled program would execute —although it must be noted that the majority of code critics ultimately disagreed with this distinction (Cayley, 2002; Cramer, 2005; Marino, 2006). For scholars like Chun and Hayles, the critical study of code offers an important avenue for engaging with digital technologies as means of creative invention that, nonetheless, reflect systems of constraint and control upon the ranges of potential outcomes (texts, performances, and other acts) that could be expressed through specific code texts.

1.3.1: Codework.

Not entirely distinct from critical code studies, but neither always focused on critical inquiry to the same degree that critical code scholars operate, is the critically artistic field of "codework." Codework examines the explicit and implicit values and meanings that might be communicated through code composed specifically as readable texts for both human and technological audiences. The most well-known example of codework would be the genre of Perl poetry, the creation of lyric poems written in the Perl computer language that double as functional code programs or scripts. Most codework texts, however, do not possess equilibrium in this sense; instead, they are generally either primarily compilation-friendly, executable code texts that approach possessing meaning to human readers *or* natural language texts whose appearance approximates the syntax and vocabulary of software code.

Scholars and artists who create and critique codework often emphasize the unique qualities of code as a form of *writing,* with all the nuances of meaning, stylistic concerns, and questions of audience that accompany it. Raley (2002) and Sondheim (2001) each praised the artistic possibilities in experimenting with code-based forms of writing that extend beyond conventional constraints of alphabetic writing. The mezangelle work of Mary-Ann Breeze (as "mez") has provided one clear style of codework in which code-like punctuation and syntax serve to imitate the meaningful construction of code but do not reflect how code actually functions. In contrast, Cayley (2002) has cautioned against criticisms of code-like texts that do not focus explicitly on code and its properties; specifically, he noted that the "code" in most codework "has ceased to function as code" that possesses only a "pretended ambiguity of address" to human and technological audiences. As a result, the focus on non-code codework texts shifts from code to something that only tangentially reflects what code is and does. Instead of looking at artificial code texts, Cayley's argument—which certainly influenced the works of Marino and Jerz—suggested an emphasized engagement with *code* in order to understand code, a perspective that reflects traditional literary studies (reading poetry helps understand poetry, novels for novels, and so on).

1.4: Digital Rhetoric

Unlike the more recently established fields of software and critical code studies, rhetorical criticism has been challenged for several decades by the increasingly shifting, amorphous nature of its disciplinary bounds. This challenge has been most prominent in recent years in regards to rhetoric's collective scholarly ability to properly and fully comprehend and explain the types of discourse present and possible in and through emerging media, many of which brought into question traditional methods of rhetorical scrutiny on the public rhetor and notable oration. This challenge is most visible in regards to the development of increasingly ubiquitous digital technologies, which caused a sort of disciplinary fragmentation as scholars either sought ways to incorporate classical models of criticism into the study of new media or assumed perspectives focusing on these new media that were distinct from traditional approaches to rhetoric.

The initial exigence in maneuvering rhetorical criticism into its current trajectory toward establishing a clear understanding of "digital rhetoric" was a need to incorporate into its folds the study of public discourse as democratic action. That is, rhetoric has developed beyond its classical focus on the single rhetor (usually a noteworthy public figure) speaking to a public audience to advocate civic change. Among the earliest significant and relevant arguments providing this influence is that of Perelman & Olbrechts-Tyteca (1969), who argued that speech and writing do *not* require radically different critical approaches to either form of communication. At roughly the same time, Burke (1969) located rhetoric "in an essential function of language itself […] the use of language as a symbolic means of inducing cooperation in beings that by nature respond to symbols" (43). By conceptualizing the production of meaning through symbols, Burke presaged the later shift for rhetorical critics to incorporate multiple modes and media through which rhetors could persuade audiences.

Several years later, rhetorical considerations begin getting mapped onto emerging technologies, most especially in regards to the computer and networks of computer technologies. McKeon (1971) argued for the promotion of rhetoric as an "architectonic art"—that is, an art producing action rather than knowledge, and which affects the performance of other arts. McKeon noted that an architectonic approach to rhetoric in a technological era was "technology itself given a rhetorical formation" (1971, p. 53). Turkle (1984) meanwhile argued that computers should be viewed as agents in their own right (pp. 271-2). These readings of rhetoric and its connection to technology offer a blurred understanding of the relationship between human rhetor and the medium through which an argument is delivered: is the computer merely a medium for discourse, or is it a separate rhetor collaborating with a human to communicate with others, humans and computers alike?

A number of rhetorical critics have explored how conventional approaches to rhetoric could effectively provide insight to communication taking place through electronic and digital media. In an effort both to distinguish "technological rhetoric" from "the rhetoric of science," to which the former is often connected and to connect criticism of technologically-focused discourse to that of civic discourse, Miller (1994) argued that there was "no conceptual vocabulary other than that of classical rhetoric [that] makes it possible to attend to the suasory nature of discourse in its full, situated complexity" (p. 93). Following this lead, Gurak (1997) examined ethos and delivery in on-line communities and how their interactions via media capable of instantaneous communication—which Gurak referred to as “unprecedented rhetorical forum[s]” thanks to their novel means of interactivity, speed of discourse, and broad scope of the base of potential participants—needed to be more clearly understood by rhetorical critics (16). For Gurak, as for Miller, the existing vocabulary of rhetorical criticism must inform further study of emerging, and especially digital, technologies.

More recently, there has been an effort by several rhetorical scholars to expand the language employed by critics so as to more fully explain and contextualize discursive practices mediated by digital technologies. For example, Fagerjord (2003) argued that rhetoric itself needs to be redefined to incorporate more successfully newer discursive media: “[rhetoric] also embraces still and moving images, visual aspects of typography (typefaces, sizes, colors, layout, etc.), and programmed procedures in computer texts. In the wide sense, then, I use rhetoric to describe how media messages are made to appear” (p. 313). This critical approach has been mostly successful through the related field of composition and rhetoric, in which scholars have experimented with the possibilities of creating and communicating meaning through digitally-produced multimodal and multimedia texts, from Kress & van Leeuwen's (2006) work with images and document layout to Johnson-Eilola's (2004) experimentation with databases and hypertexts and Carnegie's (2009) work constructing technological interfaces as forms of *exordium.*

However, many scholars have simply transferred conventional lines of inquiry to emerging environments. Warnick (2002) provided the most notable such focus by examining online texts used in the 2000 presidential campaign and ideological declarations presented in magazines about technologies. For Warnick, this was a recognizable and conventional way to learn about the ways in which particular interests are promoted, suppressed, or ignored across populations. Warnick (2007) later reemphasized the applicability of classical rhetoric to online environments, although she noted that in many ways, rhetoric as a discipline must move away from notions of single authors and works as its objects of study and toward the distinct concepts of collaborative authorships and interlinked hypertexts (p. 122). An entirely different approach was taken by Brooke (2009), who restructured the rhetorical canons in order to more clearly and directly define them in relation to the variety of new media emerging as part of developments in digital technology.

A number of scholars have offered similar approaches to reconciling traditions of rhetorical criticism with the means and ends available through digital technologies. Porter (2009) examined delivery as a means of revising existing rhetorical theory or producing new theoretical perspective, and he argued that "[t]he real value in developing a robust rhetorical theory for digital delivery lies in [productive action]" (p. 221). Porter offered a reading of data entry, transmission, and retrieval as a type of Ciceronian oratory with a specific goal, using as an example the increased availability of a patient's medical records to healthcare employees as part of their efforts to treat that patient. Zappen (2005) posited that it is necessary also to consider how the qualities of digital communication can and do affect specific types of inquiry and discourse, and specifically “the characteristics and [...] strategies of self-expression, participation, and collaboration that we now associate with [digital] spaces” (323). Zappen's argument offered rhetoric, as a field, a way to shift from a focus on modifying the vocabulary of rhetoric toward a focus on recognizing how persuasion in specific situations, for specific audiences, is fundamentally altered by the use of digital technologies. However, there have been very few publications that have attempted to identify and articulate just how persuasion could be recognized and utilized through particular technologies and media; Miller (2012) has suggested that digital rhetoric remains merely a “hypothesis—or maybe a hope” rather than an established and recognizable field of rhetorical study.

The closest that any particular initiative has come to addressing these concerns of digitally-mediated and productive rhetorical action is in the sub-field of computers and writing. Stemming from the field of composition, scholars in computers and writing have made as their focus the intersection of rhetorical invention and digital technologies, experimenting with the range of relevant possibilities available to scholars and students alike. Yancey (2004) observed that most students of writing are likely to engage regularly in types and modes of digital communication that are not addressed, either adequately or at all, by scholars and instructors of writing and rhetoric. Rieder (2010) has suggested that potential merit exists in examining particular types of code processes as rhetorical strategies for new forms of writing. McCorkle (2012) argued that rhetorical delivery operates as a form of technologically-mediated discourse involving not just invention within a particular technology but also the body of systems that facilitate a text's distribution. A small group of other scholars (Cummings, 2006; Carpenter, 2009) have observed that there is rhetorical value to be found in code as a form of writing, although there is little agreement on how best to engage it—some suggest it should complement conventional forms of writing (like a kind of frame to facilitate novel means of interaction with that conventional writing), while others question whether it might be a distinct method of communication worthy of examination and experimentation separate from other writing studies. Brooke (2009) outlined a new trivium of study for contemporary education that incorporates an “ecology of code” and its productive resources as one of the trivium's components (p. 48). Mateas (2005), Vee (2010), and Berry (2011) each outlined a computationally-oriented literacy that would describe the sets of knowledge and technical skill informing one's ability to practice code-based writing from a rhetorically aware perspective—for Mateas, this was a “procedural literacy;” for Vee, it was best dubbed “proceduracy;” for Berry, it was “iteracy.” While each of these approaches acknowledges the current problems with rhetorical scholarship of digital modes of communication, none assumes as its focus the critical examination of just how rhetorically significant meaning-making occurs in and through code.

Despite the fact that there is a significant sense of anxiety possessed by many rhetorical critics when it comes to engaging digital media beyond the scope of user-level interfaces, rhetoric as a field is nonetheless ripe for extension into the space I propose for a rhetorical code studies. Rhetorical critics are clearly interested and involved in work relating to how meaningful communication occurs in digital contexts; what is lacking is a concentrated engagement with how the technologies underlying those contexts enable, augment, and constrain meaning-making with and through digital technologies.

1.5: Locating and Occupying a Space for Rhetorical Code Studies

The field of rhetorical code studies exists within the territory I have begun to locate above. It can be recognized more clearly by outlining the foci of these related and aforementioned disciplines. Where rhetorical code studies would be most valuable to rhetoric, software studies, and critical code studies alike is in its emphasis on the rhetorical qualities and goals of computation, the underlying logic of digital technologies, at multiple levels of activity. These levels of rhetorical activity involve communication geared towards technological execution (the computation itself) and what sorts of expression that execution results in. However, the arguably most important site of activity is how computational operations are composed: the persuasive arguments suggested through procedures by developers in order to convince others that such procedures are not only useful but *optimal* in order to anticipate particular computational and expressive activities.

While meaningful communication has been addressed to some degree by software and critical code studies, and while the mechanisms and logics of digital technologies have been incorporated superficially by rhetoricians, there has not yet been a satisfactory attempt to explore the specific relationship between technological activity and development-related meaning-making at and around levels of software code. Rhetorical code studies would serve as the site of such critical efforts, and scholars from across these related disciplines could find a focus for their work in the points of intersection that connect inquiry into meaning-making, persuasion, software processes, and code-as-text.

Software scholars provide rhetorical code studies with a set of critical lenses through which to scrutinize the relationships between culture, society, and software as a guided path towards the rhetorical scrutiny of software. Of special interest are the cultural influences and constraints upon computational *logic*, as expressed in particular software paradigms and programs, that are emphasized by recent work in software studies. Cultural analysts bring to rhetorical code studies an emphasis upon the malleable, computable, and inherently *meaningful* nature of digital data separated, in many cases, from the specific situational concerns that initially generated that data. While this may initially sound alarming to rhetoricians, I argue that cultural analytics' withdrawal of data from its initial situation facilitates critical engagement with *data* and what it has the potential to mean, rather than a focus on what the data has already meant in specific contexts. Critics involved in the study of expressive processing have similarly primed the space for rhetorical code studies by drawing attention to the performative and meaningful qualities of software designed for specific ends. Just as rhetorical code studies demands an engagement with data, so too does it require an engagement with the processes that facilitate data distribution, manipulation, and analysis.

Critical code studies and codework are valuable to a rhetorical code studies through their focus on exploring the meaningful qualities of software code as meaningful text. Just as software logic can help one understand how code-based persuasion and action could occur, an examination of specific code texts and languages allow for greater insight into the specific forms and means of invention and rhetorical action that are currently attempted, and that could be attempted, by programmers for specific audiences. Critical engagement with code serves as a way to explore not just what has the potential to be created but as a way to reflect on, and to move beyond, the traditions of existing historical and contemporary code practices. For rhetorical code studies, this is significant as it aids scholars in recognizing and addressing efforts toward constructing and communicating meaning through various types of code texts.

Rhetoric, and digital rhetoric in particular, offers rhetorical code studies an established grounding in the study of meaning-making and persuasion. While software and critical code studies bring to rhetorical code studies a focused inquiry into software, code, and the logic thereof, digital rhetoric introduces into the mix a set of critical lenses and tools for investigating how individuals communicate with one another. Special attention should be paid to rhetoric's tradition of focusing on the means by which rhetorical agents attempt to persuade specific audiences for particular ends. This is a crucial quality for rhetorical code studies as it clarifies both a set of goals that developers, software users, or even technological systems work toward and the types of meaning-making they engage in in order to achieve those goals.

In order to pursue this set of amalgamated disciplinary ends appropriated from rhetoric and the critical study of software and code, this project articulates several concerns key to realizing a rhetorical code studies. First, Chapter 2 serves as an inquiry into the rhetorical nature of computation with a specific emphasis on software code, situating the processes and practices of programming within the realms of rhetoric as well as within those of the studies of software and code. While it is often viewed as existing outside the realm of persuasive influence, software development is a highly rhetorical act, and this chapter will highlight some of the fundamentally rhetorical concepts of code and how it facilitates and constrains action within, and without, the bounds of human activity. Beginning with an examination of algorithm as rhetorical procedure, the chapter connects together humanistic practices that use algorithmic logic to engage with significant objects of inquiry, both before and after the advent of the computer. Ultimately, Chapter 2 leads to an articulation of algorithmic persuasion, the use of computation in order to facilitate meaning-making in and through technological systems and interfaces.

Chapter 3 focuses on the more conventional forms of discourse that pervade and surround software code. Just as it would be inaccurate to claim that code as a form of communication is devoid of rhetorical action, it is also inaccurate to suggest that code is entirely *separate* from other types of rhetorical communication. Instead, code is infused with all manner of discourse: comments between lines of code offer insight from one developer to others about how to deal (whether productively or critically) with improvement of a particular program; email listservs highlight the social nuances of particular communities of developers, bug testers, and other users; even the websites promoting particular software programs suggest certain ways of using, reading, and writing code. Chapter 3 builds upon the previous chapter in order to demonstrate how practitioners of code acknowledge, and make use of, the agency possessed by software code within digital contexts for meaningful ends.

Chapter 4 builds upon these intertwining qualities of persuasion and procedure to offer a rhetorical analysis of a particular code artifact and the more conventional forms of communication that surround it. The artifact to be examined is the Mozilla Firefox web browser, a collaboratively-constructed open source program whose open development has spanned the last fourteen years. Mozilla was chosen in part because of its global popularity and thus its recognizability: it is a program widely used by many individuals whether they are involved in development or not. The browser has served as the site of tens of thousands of code contributions made by hundreds of programmers, where differences in stylistic framing of functions, hierarchical relationships between variables and files, trajectories for programmatic innovation, and even the acceptable amount of commented profanity have all served to highlight the fundamentally rhetorical nature of software programming as a set of highly suasive activities. At the same time, development of the Firefox browser is undertaken by volunteers who are connected to numerous proprietary organizations and their respective software projects, influencing the development of particular procedural styles of code prevalent in the program in ways that a purely "amateur" effort would likely not possess.

Accordingly, the code texts and practices shared among community members comprise an object of study that affords insight into both voluntary OSS and closed source (or proprietary) community communication and persuasion. The sheer volume of discursive and non-discursive activity that has been undertaken by developers involved in the Firefox community for over a decade means that we have the potential to see a decently representative cross-section of the kinds of rhetorical appeals made by developers in other

OSS communities (and, arguably, many non-OSS communities, but such code is more difficult to examine due to commercial and intellectual property interests). This is not necessarily to suggest that such development is exhaustively microcosmic but that the popularity of Firefox has drawn together numerous types of developers with varying skills, sets of knowledge, and levels of interest in building the program. By exploring not just the code but the discourse surrounding the code development of the browser, we can gain a clearer picture of how Mozilla Firefox, as exemplar of software code, has been designed as a rhetorically meaningful program and set of code processes.

The overall goal of this project is not simply to make the argument that code is rhetorical, but to demonstrate that approaching code and software as valid and valuable forms of meaning-making serves the interests of scholars and practitioners of rhetoric, software, and of code. In particular, by engaging code as a valid and significant object of rhetorical study, scholars can gain a fuller understanding of the means by which suasive action is achieved in an increasingly ubiquitous digital environment. Further, practitioners of code could more fully integrate rhetorically-informed concerns into their efforts at creating technological interfaces and possibilities for interaction. Access is becoming increasingly more affordable and understandable to the general public, not just to digital technologies but to the means by which to create, modify, and manipulate those technologies. Accordingly, the need to comprehend and practice critically rhetorical practices with and through code cannot be ignored or approached obliquely. Rhetoric and other disciplines in the humanities offer a set of valuable perspectives on digital technologies and their literacies that practitioners could add to the broader civic conversation: we should neither seek to distance ourselves from these available means of persuasion nor set the responsibility for critical and practical understanding of those means to others uninterested in meaningful communication.

CHAPTER 2: ALGORITHMIC PROCEDURE AND THE HUMANITIES

Among the concepts central to rhetorical code studies is that of the *algorithm*. While Chapter 1 provided an overview of rhetorical code studies as an extension of rhetoric, software, and critical code studies, here I turn from an overview of those fields to a specific engagement with the dynamics of the algorithm in relation to critical, rhetorically-oriented inquiry. The reason I make this turn is to demonstrate how algorithmic procedures are applied to humanistic scholarship in general and to rhetoric in particular. Specifically, I trace a path from the origins of the algorithm through its adoption from mathematics by computer science and engineering to its role in the critical work of humanities research. After this brief history of the algorithm and its connection to humanistic work, I discuss how the algorithm plays a central part in persuasive activities (i.e., rhetorical acts). These activities will be viewed both from a human-centric perspective and from what scholars like Hayles (2012) have referred to as a "technogenetic" perspective, which is one that identifies the interrelated co-development (or, for Hayles, co-evolution) of human and technological entities (p. 10). This concept of algorithmic persuasion, as viewed through these perspectives, works to facilitate meaningful action in a variety of digital contexts, and it serves as the focal point for rhetorical code studies.

2.1: From Algorithm to Algorithmic Culture

An algorithm is, in broad terms, a procedural framework for accomplishing a particular task. Its most common usage occurs in engineering, computer science, and mathematics as a procedure with a discrete number of tasks whose steps make use of clearly defined conditions that impact subsequent decisions within the procedure. The algorithm as a concept has its origins in the mathematical writing of Abu Abdullah Mohammed ibn Musa al-Khwarizmi, a ninth-century Persian mathematician whose work is commonly considered to have served as the basis for today's algebra (*al jabr*). In fact, the word algorithm is also generally said to be a reference to al-Khwarizmi's name (Hillis, 1998). However, as noted by Steiner (2012), the algorithm *as a concept* predates al-Khwarizmi's work, or formal mathematics in general, by several millennia. Not only did Steiner observe that the algorithm existed long before al-Khwarizmi more explicitly described it as a type of procedure, but he also argued that algorithms have played significant roles in cultural activities for most, if not the entirety, of their existence:

> Humans have been devising, altering, and sharing algorithms for thousands of years, long before the term itself ever surfaced. Algorithms needn't involve graduate school math or even math at all. The Babylonians employed algorithms in matters of law; ancient teachers of Latin checked grammar using algorithms; doctors have long relied on algorithms to assign prognoses; and countless numbers of men from all corners of the earth have used them in an attempt to predict the future. (Steiner, 2012, p. 54)

Although the idea of algorithmic procedure has been a part of human culture and behavior long before the ninth century C.E., it is through al-Khwarizmi's writing that the algorithm becomes codified as a procedural framework whose functionality is articulated through a specific grammar—initially, what would later come to be called algebra.

For al-Khwarizmi and for mathematicians since, the algorithm was the procedure through which a mathematical equation would be calculated. By constructing a framework to which a mathematician could adhere in order to solve problems, the capabilities of symbolic systems to reflect logical procedures were clearly articulated. One example of al-Khwarizmi's algebraic algorithm demonstrated the ability for a mathematician to determine the value of a particular squared number: "[if 'f]ive squares are equal to eighty;' then one square is equal to one-fifth of eighty, which is sixteen" (1831, p. 7). In mathematical notation, this equation can be demonstrated in the following steps:

Step 1: $5x^2 = 80$

Step 2: $x^2 = 80/5$

Step 3: $x^2 = 16$

The procedure to determine the value of x^2 involves condensing as many relevant factors of the equation as possible so as to calculate the value of the number to which x^2 is equal. While the notation by which algorithms could be most efficiently expressed was not developed until several centuries after al-Khwarizmi, the potential for algorithmic power had been established.

This power has become most obviously demonstrated through the application of algorithms for computational ends, thanks to the rise of computers and the scientific and engineering disciplines dedicated to their study and development. Computers operate on a form of logic known as Boolean logic, after the nineteenth century logician George Boole, who attempted to replicate the patterns of human thought through the logic of mathematical algorithms (Hillis, 1998). There are only a few fundamental concepts that drive Boolean logic, the most notable being the binary states of “true” and “false” (or “on” and “off,” or “1” and “0”). Shannon (1937) demonstrated how electrical circuits, by being opened or closed, could serve as a real application of Boolean logic. Shannon's work was used as the basis for programming machines to perform mathematical calculations, which in turn set the stage for the development of current computer technology.

By checking the state of one or more given elements within a computational system, what is referred to as “input,” the steps of a Boolean algorithm can allow a software program to perform particular tasks so as to compute and express a relevant “output.” Hillis (1998) has described algorithmic procedure and the computation it enables as being “all about performing tasks that seem to be complex (like winning a game of tic-tac-toe) by breaking them down into simple operations (like closing a switch)” (p. 4). In other words, computer science can make significant use of Boolean-powered algorithms because algorithms—especially procedures that can be *automated* by a computational system—align very effectively with the Boolean foundation upon which computers and computer data work.

Due in no small part to the increasing ubiquity and status of computer technology in contemporary society, the algorithm has become a significant concept for popular culture as well as for science and mathematics. Berlinski's (2000) *The Advent of the Algorithm*, MacCormick's (2011) *Nine Algorithms That Changed the Future*, and Steiner's (2012) *Automate This: How Algorithms Came to Rule Our World* all explicitly identify the algorithm as a paradigm-shifting phenomenon whose importance has world-changing effects. Steiner's argument in *Automate This*, for example, focused on the increasing control that complex computer algorithms possess in contemporary culture. Steiner centered his examination on the consequences of a Wall Street stock market glitch in 2010, pointing out how the "inexplicable" fiasco had the potential to wreak havoc on multiple nation's economies. For Steiner, this sense of control is critical; he ended his book with a discussion of how those who can create, understand, and manipulate complex algorithms with digital technologies are the new ruling class. This line of argument was so significant for Steiner that it comprised the title of his final chapter: "The Future Belongs to the Algorithms and Their Creators" (p. 212). Across these texts, there is a shared central premise that algorithms, especially those meant to be expressed via computer technology, are quickly gaining a prominent role at the heart of the networks and systems that power society. This prominence extends far beyond the significance of the culturally relevant algorithms that have persisted for centuries (e.g., those used in medicine, law, etc.). To be aware of algorithmic procedure or to be capable of working *with* algorithms is to be capable of influencing the trajectory of social, cultural, and political development to extents far beyond those possible by individuals who are unaware or uninvolved with the construction and modification of algorithms.

While the vast majority of academic and professional discourse related to algorithms —overwhelmingly taking place in mathematics, computer science, and engineering circles— has focused on computational algorithms, the specific constraints of an algorithmic procedure, as it is conceived of and used, are somewhat flexible from discipline to discipline; the specific field in which a scholar or practitioner works has some influence upon how he or she understands the algorithm's potential. This is important as the particular language used by a scholar or practitioner discussing what an algorithm *is* and *does* illuminates the recognized potential that scholar, or his or her discipline, attributes to algorithmic procedure.

Computer science, for example, bases its definition(s) of the algorithm on the logic of the precise and discrete mathematical models that serve as the foundation for the discipline. Brassard & Bratley (1996) defined an algorithm as “a set of rules for carrying out some calculation, either by hand or, more usually, on a machine” (p. 1). For Hillis (1998), a computer engineer, it is “a fail-safe procedure guaranteed to achieve a specific goal” (p. 78). Black (2007), a computer scientist at the National Institute of Standards and Technology, defined the term as “a computable set of steps to achieve a desired result.” However, not all scholars in the field describe algorithms in such concrete, discipline-specific terms; Edmonds (2008), for example, has stated that

> [a]n *algorithm* is a step-by-step procedure which, starting with an input instance, produces a suitable output. It is described at the level of detail and abstraction best

> suited to the human audience that must understand it. In contrast, *code* is an implementation of an algorithm that can be executed by a computer. (p. 1)

Edmonds is quick to separate algorithm as concept from the code-centric applications of algorithmic procedure for computer science, organizing the latter within the hierarchy of the former: as a result, the possibilities of the algorithm beyond the scope of computer science become much more situational and flexible.

Other fields with less direct foundation in mathematics often hold the algorithm to possess these more fluid qualities than those relevant to computer science. Accordingly, the means by which algorithms are approached and used for ends in these fields are quite different from those of disciplines in science, mathematics, and engineering. Ramsay (2011), a literary critic whose algorithmic methodology will be explored later in this chapter, described the algorithm as a concept in relatively flexible terms as “a method for solving some problem” (p. 18). Galloway (2007) qualified a specific application of algorithmic procedure for game studies as “a process with spontaneous origins but deliberate ends” (p. 12). Each of these cases brings to mind Steiner's (2012) brief historical overview of the algorithm quoted above as it relates to law, medicine, grammar, and efforts toward predicting the future. Through these definitions of algorithm as procedure *for purposefully solving problems*, non-scientific disciplines bring to the fore an emphasis on how humans approach accomplishing particular tasks. In order to clarify what the significance of this conceptual shift means for the study of algorithms within a rhetorical code studies, I will now situate the applications of algorithmic procedure across fields within the humanities.

2.2: Algorithmic Criticism in the Humanities

An emphasis on algorithms as problem-solving procedures defined broadly, rather than as mathematically-oriented tools constructed specifically for calculation, implicitly incorporates into its scope the computational algorithms that computer scientists develop. However, this type of inquiry is focused more on how a particular algorithmic procedure reflects the goals and values of its developer than on the inherent mechanical efficiency with which an algorithm can compute its input and express its output accordingly. Algorithmic inquiry in the humanities similarly emphasizes the means by which procedures facilitate meaning-making and novel approaches to critical engagement rather than the technical expertise that conventionally accompanies, or precedes, particular forms and applications of computation. Such inquiry explores both the situations that algorithms impact *and* the situations in which certain algorithms are composed, the latter of which involves the how those compositions are structured in order to make a particular case. This combination incorporates into its fold multiple scales of rhetorical activity, from the exigence that spurs a given persuasive code-based effort to the specific devices used to frame and describe the response to that exigence.

Even though my focus here ultimately turns to consider how algorithms are useful and significant components of persuasion, especially in regards to contemporary rhetorical activities, I will first outline how algorithmic procedure is used in multiple fields within the humanities. The relationship between algorithm and humanistic production has existed for millennia, from the rhetorical *enthymeme* to digitally-mediated manipulations of massive sets of data. The span of this relationship is important to recognize: algorithmic procedure has been an integral part of human thought and activity long before computer technologies were prevalent, and the ubiquity of those technologies means that current approaches to, and applications of, algorithms can be even more varied, nuanced, and complex.

2.2.1: Humanistic algorithms before, or without, computers.

While algorithmic procedure is most commonly connected to activities and experiences dealing with computer technologies, algorithms have been an integral part of humanities scholarship (and rhetoric in particular) long before personal computers became accessible and affordable to a significant percent of the population. More specifically, the idea of procedure as a means of generating or facilitating action has accompanied creative and critical practices since early uses of mnemonic devices in order to recite oral poetry successfully. As I will demonstrate here, the application of algorithms for humanistic—and especially rhetorical—purposes has been a significant component of productive and critical activity alike. This identification of an algorithmic humanities is integral to understanding how algorithms work for persuasive ends, a subject which will be taken up later in this chapter.

2.2.1.1: Enthymeme as algorithm.

The algorithm most central to rhetoric is the *enthymeme*, a concept that has predominately been defined as an incomplete logical procedure that demands work on the part of the audience in order for it to be properly and effectively expressed (Bitzer, 1959;

Walker, 1994). Specifically, the enthymeme is a rhetorically-oriented *syllogism*, a pair of premises (one major, one minor) that together lead to some particular type of conclusion or result. For a complete syllogism, this conclusion is certain, as it is explicitly stated; for an enthymeme, this conclusion becomes *probable* in that its statement remains hanging in the air for audiences to identify on their own. The most well-known syllogism includes the following components:

1. All humans are mortal. (Major premise)
2. Socrates is a human. (Minor premise)
3. Socrates is mortal. (Conclusion)

This syllogism works categorically, meaning that it defines Socrates based on the categorical descriptions into which he fits (All *A* are *B*; *C* is an *A*; therefore, *C* is *B*). More complex syllogisms can be constructed in order to create disjunctive or conditional reasoning. For example, the following is a disjunctive syllogism: “We will meet either in Paul's office or in the conference room. We are not meeting in the conference room. Therefore, we are meeting in Paul's office.” Conditional syllogisms generally include a determination of a condition being met, such as “If it is night-time...” as one of its premises. These distinctions in how syllogisms can be constructed are crucial for algorithmic logic, since the variety of arguments made possible through categorical, conditional, and disjunctive reasoning *all* work to dramatically increase the flexibility with which one could frame a particular case logically and rhetorically.

A syllogism may be chained together with other syllogisms to create a *polysyllogism*, a more complex and nuanced line of reasoning than any of its individual components might present on its own. Carroll (1973) demonstrated a number of polysyllogisms as puzzles to be solved (what a rhetorician would claim as enthymemes to be completed) in his classic *Symbolic Logic*; one such example is presented here:

(1) All writers, who understand human nature, are clever;

(2) No one is a true poet unless he can stir the hearts of men;

(3) Shakespeare wrote “Hamlet”; [sic]

(4) No writer, who does not understand human nature, can stir the hearts of men;

(5) None but a true poet could have written “Hamlet.” [sic] (Carroll, 1973, p. 170)

To complete the polysyllogism, one would ultimately need to reason that Shakespeare was clever. More fully, the deduction would recognize that a true poet is clever, and Shakespeare is argued here to be a true poet. The deductive process for Carroll's polysyllogistic example is somewhat less direct or linear than a simpler syllogism, but its computational nature is indisputable. Indeed, many contemporary algorithms rely heavily on polysyllogistic reasoning in order to compute data dynamically in numerous iterations.

In contrast to a fully articulated syllogism, the enthymeme functions as an algorithm by transforming the explicit reference to one of the syllogism's components into an implicit request that an audience compute the value of that missing component. Walker (1994) has even hinted at the enthymeme as a kind of algorithm in his description of the enthymeme as the center of “argumentative or suasory procedure” (p. 61). Similarly, Walton (2001) argued that the enthymeme suggests a “plausibilistic script-based reasoning” commonly studied in regards to artificial intelligence (p. 93). One such example of the enthymematic algorithm is the following: all men are mortal; thus, Socrates is mortal. The previous statement forces the reader to make an internal logical leap in order to discern that Socrates is mortal *because he is a human*. The “Socrates” syllogism could be rephrased by using any two of its three components; for example, Socrates is a human and therefore is mortal. The reader's ability to follow this argument hinges on the ability to express the implicit association that Socrates's mortality is dependent on his humanity *because all humans are mortal*. In other words, the enthymeme provides an opportunity for a rhetor to directly engage an audience in the expression of an algorithm for rhetorical ends. Specifically, the computational operation—the completion of the syllogism—is constructed in such a way as to convince the audience how best to calculate the remaining variables. For developers, the enthymemetic analogy can be extended to *types* of procedures in order to suggest how to solve other sorts of computation in similar manners. In this sense, the algorithm works beyond the constricted sense of “a method for solving some problem” (Ramsay, 2011, p. 18) and instead demonstrates its fundamental flexibility and contingent nature as a process through which an audience is led to convince itself to achieve action, via such means as deliberation and conditional deduction.

One trend that aids the emergence of rhetorical code studies is the recent exploration by several rhetorical scholars of the bounds of enthymematic persuasion. Specifically, they have examined whether the enthymeme can function as a rhetorical tool with value beyond the scope of conventional discourse. For instance, Smith (2004) demonstrated how visual arguments make use of enthymematic principles of probability and implicit syllogistic completion in order to persuade viewing audiences. Walton (2001) identified the enthymeme as an integral component of artificial intelligence research, tying together technological (and technologically mediated) modes of "thought" (i.e. reasoning) and communication.

Conceptually speaking, at the heart of the enthymeme is a recognition of computationally algorithmic procedures as inherently interrelated with this central form of rhetorical reasoning. The outcome for an enthymematic algorithm is expressed by a collaborative effort on the part of both rhetor, who initially provides an enthymeme as part of his or her argument, and audience, who completes its logic as part of an engagement with that argument. While algorithms in technological contexts may seem less enthymematic than mechanical, they nonetheless require the contingent interpretation of input-to-expression in order for its subsequent action to successfully communicate its meaning.

2.2.1.2: Bitzer's rhetorical situation.

Another of the most significant algorithmic frameworks related to the study of rhetoric is that of the "rhetorical situation" as described by Bitzer (1968). For Bitzer and for rhetoricians since, the concept of the rhetorical *situation* has served to comprise several integral and interrelated components of an effective rhetorical activity. First, a rhetor identifies a relevant *exigence* that he or she wishes to engage. An exigence is, as Bitzer has defined it, "an imperfection marked by urgency […] something waiting to be done" (1968, p. 6). In other words, the exigence is the catalyst for rhetorical, and any subsequent, action.

Further, a rhetorical exigence requires the capability for change to occur—a problem that cannot be avoided or resolved exists outside the bounds of rhetorical intervention. It is only once this initial variable of the situation is established that the algorithmic quality of the rhetorical situation can begin to be processed.

The second central component of the rhetorical situation is the *audience*, the body of individuals that a rhetor hopes to induce to action through his or her persuasive efforts, defined as agents "who are capable of being influenced by discourse and of being mediators of change" (Bitzer, 1968, p. 8). Jasinski (2001) observed that a rhetorical audience consists of both the audience engaging in the anticipated action and the audience "capable of influencing those with final decision-making authority" (p. 515). In order for a rhetor to successfully communicate his or her message and affect the exigence that fuels his or her message, there must be an evaluation of the audience in regards to how its members can understand and respond to the rhetor's call to action. This evaluation, which is not without its critics (discussed shortly), generally includes a recognition of the audience's values, background, and ideological leanings as well as a recognition of the modes of communication most likely to persuade the intended audience. For code, as with other forms of communication, such an evaluation relies upon recognizing when and how a particular approach could be appropriately effective, not that it is the objectively most superior means of persuading an audience.

The third component of Bitzer's rhetorical situation is a concept central to the aforementioned audience evaluation: *constraint*, the limitations and influences exerted upon rhetor and audience alike as part of the expression of a rhetorical activity. For Jasinski (2001), constraints “are circumstances that interfere with, or get in the way of, an advocate's ability to respond to an exigence” (p. 516). Constraints within a rhetorical situation include any and all restrictions upon the “available means of persuasion” for use by the rhetor in persuading the audience, from modes of communication to particular persuasive strategies or even individual choices of wording decided upon by the rhetor. At the same time, constraints comprise the range of potential means that *are* available and that the rhetor recognizes; Wysocki (2005) argued that “unavailable” components of a rhetorical situation cannot really be considered in regards to constraint if they are never really contemplated as serious possibilities in the first place. Wysocki's example consisted of a graduate student submitting a term paper written in crayon: while it could be done, no student or faculty member would accept such a break from convention.

Once an exigence has spurred a rhetor to act, and once that rhetor has identified the intended audience and the set of constraints framing his or her audience's potential reception of the provided argument, what next? It is at this point that the algorithm is expressed; the rhetor makes use of his or her chosen variables to bring about the intended outcome, i.e. the change sought in regards to the initial, urging exigence. Jasinski (2001) has suggested that even the *definition* of the relevant situation is itself a sort of exigence to be transformed through its identification. The range of possible responses to a given rhetorical act is highly contingent on the relevant exigence, audience, and constraints; however, it is impossible to guarantee that a particular audience will always respond in the same way to a given rhetor and argument. It is in this space for chaos—at least in comparison with an always predictable procedure—that the algorithmic character of the rhetorical situation becomes most intriguing. The rhetor anticipates how each variable of the rhetorical situation influences the others, but ultimately remains one more influence upon the situation. It is only when the rhetor attempts to *express* his or her argument through the situational algorithm that action can be achieved. In effect, considering the rhetorical situation as an algorithm transforms all rhetorical activity into iterations of that algorithm. Whether consciously or otherwise, a rhetor computes the procedure that emerges when *enough* of the situational variables have been identified in order to achieve a particular goal.

Admittedly, Bitzer's definition of the rhetorical situation has been challenged by a number of rhetoricians who see myriad complications with the limited variables and procedural steps identified initially by Bitzer. A common criticism centers on Bitzer's suggestion that the components of a rhetorical situation are objective and capable of observation and quantification, even though most circumstances are more dynamic than Bitzer's position would allow; for example, Vatz (1968) countered Bitzer's position by arguing that a rhetor creates a situation through the choices he or she makes in deciding to engage in a particular activity. Of special note in regards to this line of criticism is Edbauer (2005), who observed that the rhetorical situation as defined is relatively reductive, as numerous situations interrelate at any time in a larger rhetorical *ecology*. Edbauer's critique is significant for rhetorical code studies in that it draws greater attention to the *complexity* of the algorithmic procedures that make up rhetorical communication. In other words, while it is important to recognize how rhetor, audience, constraint, and exigence influence one another in an individual situation, it is just as important to acknowledge that these dynamics are themselves components of an even greater rhetorical algorithm affecting broader discursive trends.

The flexibility of the rhetorical situation (or ecology) is integral to an understanding of code-based action as rhetorical in that it points to the indeterminacy that exists between specific computational operations and the *potential* for those operations to facilitate ranges of activity in what they do (i.e. how computation results in subsequent action), how they are constructed, and what they suggest about inventing similar operations for *other* computational purposes. While the specific steps of a given procedure are generally thought of as discrete, the possibilities they suggest—and the situational concerns they address and create—are inherently complex and indeterminate, and discussions of the rhetorical qualities of procedure necessitate this recognition.

2.2.1.3: The Oulipo.

While algorithms for rhetorical purposes remain my primary focus here, algorithmic approaches to scholarship and creative production have gained traction in other fields within the humanities. For practices of literary composition-as-procedure, the group of mathematicians and writers in the mid-twentieth-century, who went by the name Oulipo (an acronym for the French name ***Ou****vroir de* ***Li****ttérature* ***Po****tentielle*, or "Workshop for Potential Literature"), may provide the clearest and most thorough insight on how creative works may be generated from the constraints of algorithmic structures. The members of the Oulipo recognized that “[m]athematics—particularly the abstract structures of contemporary mathematics—proposes thousands of possibilities for exploration, both algebraically (recourse to new laws of composition) and topologically (considerations of textual contiguity, openness and closure)” (Le Lionnais, 2007, p. 27). That is, the Oulipo acknowledged the possibilities of flexible and contingent meaning-making within the realm of mathematical computation, specifically how it could be used to influence the construction of rhetorical texts in natural languages. As noted by Queneau (2007), the objective of the Oulipo was “[t]o propose new 'structures' to writers, mathematical in nature, or to invent new artificial or mechanical procedures that will contribute to literary activity: props for inspiration as it were, or rather, in a way, aids for creativity” (p. 51). The composition of *structures*, complete with rules and constraints for successful invention within those structures, is significant because it emphasizes the possibilities of procedural expressions rather than the skill with which a particular expression has been created so as to reflect the nuances of a specific situation.

The explicit play with algorithmic procedure by the Oulipo was embraced because, according to Bénabou (2007), “[i]f one grants that all writing […] has its autonomy, its coherence, it must be admitted that writing under constraint is superior to other forms insofar as it freely furnishes its own code” (p. 41). It is not so much that writing within a set of constraints produces texts of a higher quality, but that a text *recognized to be* written under constraint(s) is superior because its author(s) and readers alike can appreciate how its code—that is, its algorithmic procedure—has been expressed in order to produce the text itself. This is not to suggest that the limits of procedural constraint (the amount and range of expressions that could be produced) is necessarily *restrictive*; as observed by Berge (2007), the early hypertext “A Story as You Like It” by Queneau, a founding member of the Oulipo, suggests numerous potential reading “paths” through the narrative (as the reader makes decisions about how the story will develop). Similarly, Queneau's *Cent Mille Millardes de poèmes* (“one hundred thousand billion poems”), a collection of ten sonnets that all share the same rhyme scheme, offers the reader 10^{14}, or 100,000,000,000,000, potential poems that could be constructed by the act of swapping in or out a given line from one of the sonnets with another, e.g. for sonnets A-J, one could use line 1 from sonnet A, line 2 from sonnet D, line 3 from sonnet F, line 4 from sonnet A, etc. (Berge, 2007, p. 118-121). While the members of the Oulipo explored these structures to demonstrate the possibilities of invention through procedural constraint, it may also be accurate to say that they highlighted the existing means of rhetorical invention and arrangement and offered new insight by drawing attention to the procedural structures that underpin decisions surrounding particular rhetorical situations. To repeat Ramsay's (2011) definition from earlier in this chapter, the algorithm is complicated far beyond its general definition as a “method for solving some problem” (p. 18). In particular, the Oulipian algorithms function rhetorically in that the *potential* for meaning-making is emphasized through the literary structures developed by the Oulipo.

2.2.1.4: Magic as algorithm.

Beyond literature, as stated earlier, algorithms have been employed for cultural ends for centuries. Among the most powerful of these ends has been to demonstrate the transformative capabilities of algorithmic procedures to effect action in seemingly inexplicable ways. Cramer (2005) has argued that the algorithm, when applied outside the bounds of mathematics for much of history, has been articulated as *magic* due to its general lack of easily-understood structure and means of expression. Specifically, Cramer stated that "[t]he technical principle of magic, controlling matter through manipulation of symbols, is the technical principle of computer software as well" (2005, p. 15). This shared quality is significant for Cramer because the notion of computational expression as action can be identified in systems central to culture from religion to poetry. Most integral to the quality of manipulation *through symbols*, to use Cramer's terms, is the fact that language itself becomes the means of algorithmic expression. That is, meaningful *communication* from one or more rhetors to a particular audience facilitates meaningful and humanistic *action*.

Cramer (2005) demonstrated this relationship between communication and action as an integral facet of cultural expression, using traditions of religious mysticism in Judaism and Christianity as examples of applied algorithmic procedure that paved the way for current digitally-oriented approaches to computation. Specifically, Cramer associated contemporary databases and complex algorithms with mystical efforts such as those of Kabbalists geared toward "divine creation through the computations of the letters of God's name" (2005, p. 35). While the goal of most current algorithmic expression in software is not aligned with mysticism and religious revelation, Cramer's point is nonetheless made: algorithms play a crucial role in significant social and cultural systems, and the output they generate is meant to have just as significant an impact on the structures they affect.

For Cramer, the use of language as a way to express action through the manipulation of symbols *and the ideas they represent* is especially important for the art of rhetoric. Notably, Cramer described the rhetorical canon of memory—and the classical approach to memory through the use of a "memory palace" through which a rhetor navigates through his or her argument—as an algorithm to be expressed through the act of rhetorical delivery. Cramer argued that, "[w]hile memoria involves no algorithmic computations in itself, it could be regarded as the first implementation of a visual user interface for alphanumeric codes" (2005, p. 43). In other words, the ability of a rhetor to conceptualize his or her action through language (whether in oral, written, or some other form) facilitates that rhetor's performance of the algorithmic computations comprising his or her argument when communicating it to a particular audience.

2.2.2: Humanistic algorithms with, or since the advent of, computers.

While algorithmic procedure and humanistic activity have been intertwined for much of human history, it is crucial to point out that computer technology has radically transformed this relationship. The algorithm has become not simply a means for performing traditional rhetorical or critical action but also as a form of meaning-making in its own right, although this latter aspect is not yet widely recognized. In other words, the algorithmic code that comprises digital software is rhetorically powerful thanks to its ability to engage data, machine, and human alike. For most scholars, code's engagement is primarily as a tool to facilitate other experiences, but I will demonstrate how the algorithmic basis of code, used specifically for rhetorical ends, can lead us to a moment of clarity in which we reconsider how code influences us to act—a quality that can best be described as algorithmic persuasion.

2.2.2.1: Algorithmic criticism.

Possibly the most direct use of algorithmic procedure to facilitate humanities scholarship, both conventional and unconventional, is Ramsay's (2011) concept of "algorithmic criticism." For Ramsay, algorithms provide avenues for literary research and interpretation through their expressive transformations of texts. Among the means by which Ramsay demonstrated the literary potential for algorithmic procedure is the aggregating and crunching of particular types of data. For example, Ramsay compared the most frequently-used terms and ideas provided by the major characters in Woolf's *The Waves*, demonstrating how the commonalities between characters' most frequent terms allow readers to draw new connections between those characters. Included below (in Table 2.1; Ramsay, 2011, p. 13) is the set of frequently-used terms for three of the novel's six protagonists when Ramsay's algorithm has been computed. Ramsay's line of inquiry sought out the terms that were not only most frequently used *but also* not used frequently by any other character, i.e. the terms that were unique to, or primarily associated with, each individual character.

Table 2.1: Lists of term frequency from Woolf's The Waves, compiled by Ramsay (2011)

Bernard		Louis		Neville	
thinks	rabbit	mr	clerks	catullus	loads
letter	tick	western	disorder	doomed	mallet
curiosity	tooth	nile	accent	immitigable	marvel
moffat	arrive	australian	beaten	papers	shoots
final	bandaged	beast	bobbing	bookcase	squirting
important	bowled	grained	custard	bored	waits
low	brushed	thou	discord	camel	stair
simple	buzzing	wilt	eating-shop	detect	abject
canopy	complex	pitchers	england	expose	admirable
getting	concrete	steel	eyres	hubbub	ajax
hoot	deeply	attempt	four-thirty	incredible	aloud
hums	detachment	average	ham	lack	bath

One conclusion related to new connections which Ramsay is able to draw is that, although the character Louis is self-conscious of his accent (as the only Australian in the group), the other characters pay that cultural distinction little attention: no one else ever mentions the terms “western,” “Australian,” or “accent.” To them, these concepts seemingly do not matter (Ramsay, 2011, pp. 12-14). This sort of reading by Ramsay—in which previously-unrecognized meaning is exposed through algorithmic expressions—demonstrates a new form of reading as much as it does a new form of research. Ramsay argued that “[t]he critic who puts forth a 'reading' puts forth not the text, but a new text in which the data has been paraphrased, elaborated, selected, truncated, and transduced” (2011, p. 16). In other words, the emphasis for algorithmic critics should be placed as much on what and how relevant critical work occurs as on what it reveals through its expression.

Ramsay's approach to criticism clearly identifies algorithmic procedure as a valuable tool to be used for scholarly criticism. Specifically, Ramsay's use of algorithms to explore novel means of reading allows him to generate paratexts that transform a given text or set of texts into a new object of study. When employed in this manner, the algorithm is a method for interpretation that, in part, quantifies those components of an original text which a critic deems potentially significant (the text transformed into the paratext). In Ramsay's algorithmic reading of Woolf, the word frequency of several main characters enabled him to draw connections between certain characters and to highlight some of the concerns central to each character. The significance of Ramsay's algorithmic criticism is that it affords scholars a new way of engaging texts for humanities ends but does not fundamentally transform the process of criticism itself. That is, while the means of gathering data algorithmically differs drastically from other critical approaches (such as close reading), the object of inquiry remains focused on meaningful literature. The constitution of algorithmically transduced literature generated through the deformation of an initial text does not interfere with conventional scholarly work: the algorithmic critic produces his or her own text (the algorithmic paratext) to be interpreted rather than look at the initial, non-transformed text.

Related to Ramsay's approach to algorithmic criticism is the critical method referred to as *distant reading*, popularized among literary scholars by Moretti (2005). For Moretti, the traditional act of close reading undertaken by critics—in which meaning is drawn out of a given text through a focused examination of its content—is not an all-encompassing body of knowledge useful or important to the field. Distant reading, in which a large number of texts are algorithmically processed for interesting data (on a scale even beyond that of Ramsay), provides a means of emphasizing the interconnections and structures that exist across those texts (Moretti, 2005, pp. 1-2). In essence, distant reading is a form of literary criticism that is made accessible for scholars only through increasingly powerful computer technologies that can handle complex and nuanced algorithms to process data for humanistic ends.

Where Ramsay examined trends among the most frequent terms used by characters within a single Woolf novel, Moretti explored literature through more expansive means. Some examples of these types of distant reading include the popularity of certain types of novels in Britain over the course of a century (pp. 14-16) and how the "presence of clues" and their qualities in detective stories created branches of similarity between Arthur Conan Doyle and his contemporary rivals (pp. 72-74). In most of the cases he describes, Moretti's strategy for critical reading is facilitated by an algorithmic approach to interpretation—as with Ramsay, Moretti has created meaningful paratexts through his exploration of the connections between available types and ranges of data in the original texts he collected together.

2.2.2.2: Critical code studies: algorithm as communication.

In contrast to Ramsay's or Moretti's work with algorithms *as a tool or lens* for criticism is the main body of scholarship related to critical code studies, which puts at its focus code *as text*. Implicitly, this focus suggests that a given body of code, and its author(s), have something meaningful to offer to its reader, whether intentionally or not. An additional significance of focusing on code is that *the algorithm itself* becomes the object on which inquiry is centered: how it is structured, phrased, and expressed. In this light, code is no more a simple tool than any other form of language, and its capacity for meaning-making is not only acknowledged but emphasized.

It is easy to consider the semiotic value of code when its syntax closely resembles that of natural language, especially English. The idea that code could and should be written primarily for human readers has its origin in Knuth (1992), whose idea of "literate programming" required a radical reconsideration of how computer science might be approached. For Knuth, it was crucial that code be composed in such a manner that its meaning was clearly articulated to human audiences; its ability to be executed by a computer was secondary. This perspective has been echoed by influential programmers since then, with the most notable being Matsumoto (2007), who urged his audience specifically to treat code "as an essay," using the Ruby language to provide an accessible means of doing so (p. 477). Cayley (2002) experimented with the possibilities of maximizing code's ability to perform computational tasks while clearly communicating its goals to a reader in a conventionally understandable manner. Cayley described this sort of code as a "codework," highlighting its distinction from the vast majority of code that is technically productive but not intended to be artistic in form. The following is a brief excerpt from one such codework:

```
if invariance > the random of engineering and not categorical then
  put ideals + one into media
  if subversive then
    put false into subversive
  end if
```

```
    if media > instantiation then
      put one into media
    end if
  else
    put the inscription of conjunctions + one into media
  end if
```

(Cayley, 2002)

The above excerpt is from a codework text Cayley composed in HyperTalk, a programming language developed in the late 1980s for the Macintosh hypermedia program HyperCard. Its syntax closely resembles that of English, making it easily readable (at least in relation to many other programming languages). The program in which this excerpt exists works as a text generator, with the various terms Cayley included here (such as `media`, `subversive`, and `ideals`) serving as “containers” for variable data values as part of the code's expression. For example, `media` holds a number whose value that changes depending upon certain conditions (such as whether the current value of `media` is greater than the current value of `instantiation`). The meaning of the code, then, emerges not only from what the code *says* in English (or at least near to it) but what it *does* computationally, even if we might view the functional purpose of this program to be trivial. Cayley acknowledged the code's “ambiguous address” of both human reader and computer interpreter, implicitly suggesting that there existed a rhetorical situation for each of these audiences (2002). In addition, this relationship of saying and doing in code and codework mimics the relationship in written language of saying and appearing, what Lanham (1993) referred to as a bi-stable oscillation that is never eradicated but might fluctuate in proportion between audiences and contexts.

A second example of executable code whose content closely resembles natural language (i.e. English) is Wall's (1990) poem “Black Perl.” Wall was the initial creator of the Perl programming language, initially designed to facilitate the construction of Unix scripts, and “Black Perl” highlights the ambiguity that can exist between code as a set of machine operations and as meaningful discourse presented to human readers. In its entirety, the poem reads:

```
BEFOREHAND: close door, each window & exit; wait until time.
     open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
     reverse its length, write again;
     kill spiders, pop them, chop, split, kill them.
          unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
     kill them, dump qualms, shift moralities,
     values aside, each one;
          die sheep! die to reverse the system
          you accept (reject, respect);
next step,
     kill the next sacrifice, each sacrifice,
     wait, redo ritual until "all the spirits are pleased";
     do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
     exit crypt (time, times & "half a time") & close it,
     select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
     wait, wait until time;
     wait until next year, next decade;
          sleep, sleep, die yourself,
          die at last
```
(Wall, 1990)

Wall's poem was, at the time of its initial publication, a fully compilable body of code, meaning that it could be run as a program, although its output was simply for the program to exit. Since then, the poem has been revised several times, each keeping the poem's potential to be executed in line with developments in the Perl language itself. Across its iterative versions, the content has remained primarily intact, and the poem's message—of a magic-wielding programmer who can bring forms to life and exert his or her influence over them (such as popping spiders and sacrificing sheep)—maintains its supernatural ambiance. Those "in the know," i.e. programmers, had the tools with which to affect the virtual world of the computer system, while those without that technical knowledge were relegated to understand its message only as a conventional piece of poetry; this sense of knowledge as power suggests the argument made by Steiner (2012) that algorithmic manipulators will rule the world thanks to their computational abilities. Further, the computational action that Wall's poem/program induces is invisible and trivial compared to the effect it can have on human audiences: the program simply exits after running, while its potential and interpretable meaning is likely to vary between individual readers.

While Cayley's and Wall's examples are relatively readable and accessible, most code —as Marino (2006) has observed—does not resemble literature or conventional discourse. This means that this dichotomy of audiences for code-based communication may seem to skew more towards the technological than to the human. Several critical code scholars have interrogated this balance, questioning the value of a traditional human-oriented communicative hierarchy. Even though there is likely to be disagreement over the relevance of such a perspective to rhetoricians, it is important to consider the consequences of this view in order to understand how algorithmic procedure works for persuasive, and therefore rhetorical, ends.

2.3: Algorithmic Persuasion

The notion that an algorithm can, and does, work persuasively is a radical departure from the conventional definition of the algorithm. The conventional definition, embraced by mathematics and computer science, emphasizes the discrete procedural execution of an algorithm so as to situate discussions of the components that comprise it. In some computer science contexts, there is significant discussion about the optimal way to construct a particular program or function in a given language—focusing not so much on what a procedure *means to do* but rather on how to clearly, effectively, and efficiently state and structure the steps of that procedure. There is an implicit recognition of the procedure's purpose, but it does not occupy the center of discussion. As Edmonds (2008) noted, novice programmers often find themselves needing to shift mentally "from viewing an algorithm as a sequence of actions to viewing it as a sequence of snapshots of the state of the computer" (p. 6). This shift, he argued, is significant because it draws attention to how code computes data from one action to the next within a procedure rather than on what the end result does or means.

However, the idea of computation as action and algorithm as means of facilitating action is critical to an understanding of how algorithms might work persuasively. In other words, the novice mentality described by Edmonds (2008) is closer to the mark for thinking critically in regards to algorithms than the conventional "learned" mentality. Berlinski (2000) noted that an algorithm is, in addition to the strict, conventionally computational procedure he had initially provided for his reader, "a set of rules, a recipe, *a prescription for action*, a guide, a linked and controlled injunction, an aduration, a code, an effort to throw a complex verbal shawl over life's shattering chaos" (p. xvi, emphasis added). Berlinski's reference to a "prescription for action" is not just a means of defining a particular action to occur but to call for that action to be undertaken—making the algorithm not unlike a procedural engagement with a rhetorical exigence.

What a recognition of algorithm *as means of persuasion* offers rhetoricians, then, is an opportunity to explore how a seemingly "machinic" manner of discourse, i.e. the code of digital technologies and media, can provide insight into the interplay among the canons of rhetoric in order to influence the potential expression of a particular rhetorical situation. For example, how does style affect code-based composition practices in order to facilitate action in human agents? How might a critical acceptance of a technological code compiler as co-rhetor (inventor, arranger, etc.) alter our understanding of the concepts of constraint or suasion? As digital technologies become more advanced and accessibly modifiable (in the sense of code syntax reflecting natural language and performing machinic functions), these rhetorical concerns become increasingly significant: if we are to understand how we communicate with and influence one another with the aid of digital technologies, it is important to understand how we are capable of influencing particular types of suasive action and influence to occur through the use of those technologies. Much of the analysis provided in this text focuses on code as a means by which software developers persuade other developers to promote particular types of computation and anticipated end-user action. However, we should also acknowledge the significant influence of developers' rhetorical appeals upon the software activities actually experienced by users. A consideration of the rhetor's ability to affect, at one or more code levels, constraints that extend to other modes and means of action, is vastly different and distinct from traditional approaches to rhetorical use that may take as givens the technological mechanisms constraining particular discursive efforts.

2.3.1: Procedural rhetoric.

There is a *seemingly* trivial—but ultimately significant—difference between algorithmic persuasion and the related idea of procedural rhetoric. Procedural rhetoric is a concept developed by Bogost (2007) as a way to describe the influential qualities of *systems* exerted upon individuals who make use of those systems. Specifically, Bogost's discussion of procedural rhetoric did not focus on code-based algorithms explicitly as much as on interactive systematic procedures like video games and how they persuade game players to act within a particular game. For Bogost, the rhetorical capabilities of a procedural system offered new ways of engaging specific populations. For example, a video game teaches its player the rules of its "world" through activity within the game; the player, through trial and error, explicit tutorial, or both, learns what behaviors and perspectives are acceptable and "valid" while engaging that game's system. The rhetorical action that occurs at the user level emphasizes the way(s) that a game's developers intend for its content to be encountered by a player agent. However, there is also significant rhetorical action at the developer level, demonstrated by the ways through which the developers constrained particular means of user interaction with the processes of a given game world—effectively making use of another level of procedural rhetoric, wherein one developer persuades another, to facilitate the game itself. As Bogost described it, "processes define the way things work: the methods, techniques, and logics that drive the operations of systems from mechanical systems like engines to organizational systems like high schools to conceptual systems like religious faith" (2007, p. 3). This definition suggests that code might play a prominent part of Bogost's discussion of procedural rhetoric. However, Bogost noted that some processes that "might appear unexpressive, devoid of symbol manipulation, may actually found expression of a higher order" and explained how humans perceived as "breaking procedure" in reality have constructed new processes in order to complete tasks (2007, p. 5). This adjustment of emphasis (to "higher order" expression), while hinting at the possibilities of computation for rhetorical ends, facilitated Bogost's close analysis of video games and gameplay experiences as procedural expressions.

Outside the paradigm of games, this idea of procedurally structured (or generated) rhetorical activity exists classically in the form of the enthymeme, but it has been explored further by Lanham (2003), who referred to rhetorical procedures as “tacit persuasion patterns” (p. 119). For Lanham, tacit persuasion occurs constantly, since we are rarely ever acutely aware of all influences attempting to exert themselves upon us. As Lanham observed, individuals often “feel” the presence of tactic persuasion patterns “subconsciously, even if we do not bring that knowledge to self-awareness” (2003, p. 120). This description hints not just at a passive acceptance of rhetorical appeals but a subconscious engagement with the enthymemetic arguments provided by a rhetor across multiple levels of language. As a scholar interested in speech and writing, Lanham focused his scrutiny primarily on persuasive techniques available in language, such as rhyme, chiasmus, alliteration, and anaphora. Each of these devices provides a rhetor with the ability to draw or hold the attention of an audience that might otherwise have not bothered to heed an argument. While Lanham did not address the possibilities of tacit persuasion patterns or devices in artificial (code) languages, they nevertheless exist and likely already are used frequently by many developers working collaboratively.

I will take up where Bogost (and Lanham, albeit implicitly) left off in regards to the potential for expressive action through the seemingly “unexpressive” languages of computer code. Code might be described as unexpressive in that it creates and communicates meaning in ways that often differ from conventional means of discourse; however, it is precisely because of this quality that emphasizes its procedural nature. In this text, I focus primarily not on the expressive act of procedural execution (which would highlight the general user's experience with a particular software program) but instead on how the construction of algorithms in and through *code* as both text and process functions persuasively on user and developer as well as, to a lesser extent, on other technological systems. In other words, my interest in algorithms is focused on code as a means by which algorithmic procedures are articulated for rhetorical purposes.

2.3.2: Algorithms we live by: Recognizing persuasive algorithms.

In order to discuss how an algorithm can persuade, it is necessary first to recognize how algorithms in code are composed: what they do (the actions they attempt to induce), what they say (how their operations are constructed), and how they say it (what those operative constructions mean to different audiences). These qualities are not any different from conventional rhetorical concerns of invention, style, and delivery, but their construction in code might certainly make it seem as if they are. One particularly significant quality of code, as with other forms of language, is the metaphorical role that a given code operation, or set of operations, can play in making its function(s) understandable to human audiences. Lakoff & Johnson (1980) argued that “metaphor is not just a matter of language […] on the contrary, human *thought processes* are largely metaphorical” (p. 6). Here, I argue that not merely are human thought processes metaphorical, but computational processes as well; the rhetorical actions they symbolize succeed primarily because of the metaphorical ways that human audiences (developers and users alike) are influenced to understand them in particular contexts for particular purposes. At the same time, it is important to observe that writing or speaking about algorithms is itself a metaphorical activity that frames procedure as description. As Bogost (2007) has noted, that "only procedural systems like computer software actually represent process with process" (p. 14). With this in mind, I hope to identify some integral ways that code processes make meaning for developer audiences in what those processes do function-wise and in how they are structured for the sake of readability as well as for expression.

In order to make this argument, I will demonstrate how relatively simple code algorithms can appear to work, and do work, persuasively. In the following pages, I address several examples of increasing complexity that communicate at various explicit and implicit levels the meaningful action they mean to effect through the computational operations that comprise their text forms. These examples are: FizzBuzz, a program that iterates through a specific body of data; quine, a program that outputs its entire code content; and HashMap concordance, a program that tracks word frequency across the text of an inputted source (in this case, Bram Stoker's *Dracula*).

2.3.2.1: Case 1: FizzBuzz.

The first example to be scrutinized is relatively simple in construction and intent. It is the focus of a common hiring test for programmers and is referred to as "The FizzBuzz Test." The goal of this algorithm is to take the numbers one through one hundred and print them out, one at a time, *unless* they are multiples of three, in which case the word “Fizz” should be printed, *or* if they are multiples of five, in which case the word “Buzz” should be printed. There is an implicit extra rule here; specifically, numbers that are multiples of fifteen (that is, both of three and of five) should print out “FizzBuzz.” The entire output of the program—which is identical across all four of the examples provided below—is the following set of wrapping lines:

```
12Fizz4BuzzFizz78FizzBuzz11Fizz1314FizzBuzz1617Fizz19BuzzFizz2223Fiz
zBuzz26Fizz2829FizzBuzz3132Fizz34BuzzFizz3738FizzBuzz41Fizz4344FizzB
uzz4647Fizz49BuzzFizz5253FizzBuzz56Fizz5859FizzBuzz6162Fizz64BuzzFiz
z6768FizzBuzz71Fizz7374FizzBuzz7677Fizz79BuzzFizz8283FizzBuzz86Fizz8
889FizzBuzz9192Fizz94BuzzFizz9798FizzBuzz
```

Depending upon the language, and even more importantly depending upon *the way that a programmer approaches the problem*, there may be dozens of ways to construct this algorithm effectively. The path taken by a particular developer to accomplish this task provides a great deal of rhetorical insight about his or her preferences in computing data. While the immediate goal is for a particular developer to demonstrate to his or her potential employer that he or she is a competent coder and thus worthy of hiring, the developer also demonstrates more generally his or her understanding of how “best” to use a given language (and the constraints that might frame that understanding). Further, the developer also communicates through the written code how he or she *thinks* computationally through the frame of that particular language.

Table 2.2: Two example FizzBuzz loops in Javascript

```
for(var i=1;i<=100;i++) {
  if ((i%3==0) || (i%5==0)) {
    if (i%3==0) {
      console.log("Fizz");
    }
    if (i%5==0) {
      console.log("Buzz");
    }
  } else {
  console.log(i);
  }
}
```

```
for(var i=1;i<=100;i++) {
  if (i%15==0) {
  console.log("FizzBuzz");
  } else if (i%3==0) {
    console.log("Fizz");
  } else if (i%5==0) {
    console.log("Buzz");
  } else {
    console.log(i);
  }
}
```

In Table 2.2, the FizzBuzz algorithm has been written in two similar but significantly different ways using the syntax of the Javascript scripting language, a popular code language used in web pages, PDF documents, and even desktop applications. This sort of algorithm is described as a *loop* because it continues to compute results so long as the proper conditions are met—in this case, while the input amount (`i`) is a number lower than or equal to 100. The example on the left frames its computation in an initial "catch-all" condition statement, i.e. that `i` is a multiple of three *or* of five. Then, it checks each of those sub-conditions independently of one another, meaning that `i` could simultaneously be both a multiple of three and a multiple of five, triggering the operation that will execute when each of those conditions is met (printing both "Fizz" and "Buzz"). In comparison, the example on the right in Table 2.2 functions due to a logic of exclusion. First, it checks whether `i` is explicitly a multiple of fifteen. If it is not, then the loop checks first if `i` is a multiple of three. If *that* condition is not met, *then* the loop repeats its check for `i` as a multiple of five. These conditions are dependent upon one another: that is, in the code on the right, a number computed to be a multiple of three is not also computed as a multiple of five.

The variations on how to construct and express a FizzBuzz algorithm are not limited to these sorts of conditional checks, either. Table 2.3 (below) contains two examples of the algorithm as composed in the Ruby programming language.

Table 2.3: Two example FizzBuzz loops in Ruby

```ruby
for i in 1..100
  if i%3 == 0 then
    print "Fizz"
  end
  if i%5 == 0 then
    print "Buzz"
  end
  if i%3 != 0 && i%5 != 0 then
    print i
  end
end
```

```ruby
100.times do |i|
  i = i+1
  if i%15 == 0 then
    print "FizzBuzz"
  elsif i%3 == 0 then
    print "Fizz"
  elsif i%5 == 0 then
    print "Buzz"
  else
    print i
  end
end
```

The major difference between these examples lies not in how the specific conditions are constructed (although these do differ between the two) but instead in the *type of function* that is used to form the loop itself. This is significant in that there is a fundamental shift in the logical structure of the loop and, also, of the hypothetical larger program of which the FizzBuzz algorithm might be a representative part. The example on the left of Table 2.3 makes use of a `for` loop, which—as with the Java examples—iterates through a body of data and computes each item within that body before moving to the next. In these loops, that data has been the set of integers from one to one hundred, but `for` is not limited to this kind of data as its input. The example on the right, however, only imitates that kind of looping behavior. It technically repeats the operations within its scope a set number of times and, in doing so, manipulates the value of a variable (`i`) within its scope each time. Because of how the `times` method functions, counting begins at zero rather than one, and thus `i` needs to have an extra number added to its current value (the first operation within the `times` block) before the remaining operations can accurately be computed.

Conceptually and metaphorically, this distinction between `for` and `times` reflects a fundamental distinction between iteration and repetition. The `for` loop suggests to a developer, and to the machine, that similar types of data are going to be computed through a series of operations whose scope and syntax may be influenced by the specific data being computed at any given moment. In contrast, the `times` pseudo-loop suggests that it will execute *the same operations* a set number of times; any data manipulated differently from other data as a part of that loop is an incidental consequence of its code composition. More generally, each FizzBuzz example—and arguably any code that processes a body of data—is a computational metonymy: the "loop," which implies a cyclical return to its origin, is in actuality more like a corkscrew. Its abbreviated description of operations to be executed across its data parameters never truly returns back to "the beginning" of the code, as each iteration transforms the code both in how it reads and in how it operates.

While the function of the FizzBuzz algorithm, as represented by these examples, may not initially appear to be very rhetorical in nature (since it checks a set of numbers and prints out numbers or words), it serves as a concise example of the persuasive capabilities of code. Specifically, the FizzBuzz algorithm provides meaningful information to its authoring developer, to any other human readers, and to the machine related to what the author understands about how to engage in the manipulation of a set of computer data. If the FizzBuzz code in each example is read as an excerpt from a larger program, its contents signal a set of rhetorical and computational decisions that have been made about how to most effectively accomplish its task. In essence, FizzBuzz communicates more, and other, than its output: it suggests to other agents, human and technological, how the author has identified the central means (or at least *one* central means) around and through which to compute relevant data and facilitate the desired result.

2.3.2.2: Case 2: Quine.

The second example I include here is known as a *quine*, defined concisely as a self-reproducing program. In other words, the output of a quine is the sum of its code content. The term quine comes from the mid-twentieth century logician Willard Van Orman Quine, who noted the following self-referential paradox: "'Yields a falsehood when appended to its own quotation' yields a falsehood when appended to its own quotation" (Quine, 1976). The statement can be neither true nor false, and it is only in the combination of concept and quoted concept that the paradox—the deeper meaning of the statement—emerges. Rhetorically, the algorithmic quine offers an opportunity for a rhetor to consider what it means to construct a "logically sound" argument. Is it merely an appealing effort at persuasion, or is there an internally valid consistency to it? Does the former require the latter? I suggest that, yes, an algorithmic argument generates much of its appeal through its consistent logic, although this consistency does not necessarily demand any objective truth or accuracy to succeed.

One simple quine, written in the Ruby programming language, is a single-line program that reads as follows:

`x = "x = %p; puts x %% x"; puts x % x` (“Quine”, n.d.)

Within this line of code, there are two distinct operations that occur. The first is to define the variable `x`, and its value consists of the characters within the quotation marks—this operation ends at the second semicolon. In essence, `x` is defined as a “string,” an arbitrary set of alphanumeric characters. The second operation in the quine, beginning directly after that same semicolon, recalls and displays the string's content exactly, via the `puts` function. It is important to note that this second operation *must* include its own call as part of its output in order for the program to be considered fully self-referential (i.e., the recall itself has to appear in the output of the recall). When `x` is defined in part as `%p`, `%p` serves as a position in which other content might be substituted later; `puts x % x` effectively inserts the content of `x` into itself so as to print out the quine's input correctly. Many programmers separate the content of a quine into two components, what they refer to as the code (the operations to be computed when the program is run) and the data (the non-computing replication of those operations). This binary quality of the quine, for developers, allows them to make explicit note of what “runs” and what is output.

When we examine the quine rhetorically, this distinction changes somewhat as we turn from computational success to meaningful action as a criterion of evaluation. The action of the quine's code is to reveal its data; the action of the data is to highlight the means by which it was revealed—i.e, the code itself. It is significant that the two components function reflexively rather than the code unidirectionally working "on behalf of" the data; the quine as a whole is displayed as a *seemingly* complete persuasive entity. It is this relationship that Cayley (2002) referred to when he addressed code existing as both text and not-text (as objects of study); the code and data components of a program are inherently interrelated, but the reading of code as executable action and as meaningful language are two different activities. For the quine, much of its effect—its demonstration of its "completeness"—is due to the appearance of the quine as a transparent argument. In simple terms, this appearance suggests that the quine does (only) what it says and it says (only) what it does, presenting the *semblance* of a code-based chiasmus which is completed when the input is ultimately outputted.

The quine *also* implies, though, that it is merely a simple program which presents a plainly-stated message, i.e. its content. In a sense, this implication is the perfect argument that could be made by a rhetor: the tools used to make a case *comprise* that case. Miller (2010b) has pointed out the common rhetorical tactic of self-denial as a way of playing down or otherwise concealing the acknowledgment that a persuasive effort is currently taking place. A significant component of this tactic, she argued, is *mimesis*, the idea—however flawed or inaccurate in practice—that language has the potential to represent its subject so clearly and faithfully that it need not (or cannot) deceive in its representation. As a result, a rhetor must work not only to conceal his or her true intentions in using language for a given purpose but also to conceal the fact that the rhetor is working to conceal. This idea of language as apparently, but not actually, mimetic is key to an understanding of the quine and what it can achieve rhetorically. It is not *simply* the sum of its parts (that is, its expression does not only equal its content); it is a means of suggesting, in more general terms, that all code can be reduced to such a description.

One complication for the quine—and thus, by association, any executable software—is that it is possible to hide from audiences what the true intentions *and content* of a code-based program may actually be. Thompson (1984) demonstrated the ease with which a skilled developer might circumvent the transparency of code composition and execution by inserting instructions into a machine's compiler software. The compiler is an intermediary, a translation program whose function is to transform source code (readable by humans) into an executable program (readable by the machine). Thompson revealed that it was possible to manipulate the code of a compiler in such a way that it would be undetectable to anyone using the compiler, *along with any other programs the compiler subsequently compiles*, for other purposes—including the execution of a quine. Thompson's point was that one could trust *only* the code one wrote: all else was potentially devious, even a program that purported to print the entirety of its own code. While the quine itself may not be to blame for any malicious behavior on a compromised system, it is nonetheless reliant upon the ecology of technological and human agents in which it exists in order for its computation and subsequent expression to occur “successfully.”

The quine provides a valuable example of code as a text that is not significant merely because it acts rhetorically through its expression—by revealing its content—but also because it emphasizes the complicated nature of code as a text-practice that does, and says, more than what it explicitly describes in its composition and expressed performance. Just as with more conventional rhetorical activities with foci on meta-rhetorical subjects, the quine has the potential to influence audiences to reconsider the communicative event itself; in the case of the quine, this reconsideration results in an awareness of the self-description as a necessarily enthymematic, rather than a fully contained and transparent, argument.

2.3.2.3: Case 3: HashMap concordance.

The third example to be discussed transforms and deforms an existing text in order to bring to light new meaning that might not have been exposed in the text's original format. This sort of code is aligned closely with the algorithmic criticism introduced by Ramsay (2011) for literary ends. Here, the example code is included not so much to demonstrate the possibilities of code for literary criticism but instead to highlight how algorithmic manipulations of text work on different audiences rhetorically. This example, composed by Shiffman (2011) and included below, was written for the Processing integrated development environment (IDE) and uses a streamlined syntax based on the Java programming language. Shiffman's code makes use of a type of data called a `HashMap`, which serves as a way to store a "collection" of individual data elements that each has its own identifying `key`. By default, the program takes a plain text file with the contents of Bram Stoker's *Dracula* (via the electronic Project Gutenberg public domain library) and displays each word in a font size proportional to the number of times that word appears within the novel. The novel's words, reconfigured as a `HashMap`, offer in their recombined form a way to read meaning in the text based upon the frequency of particular terms and concepts. The following is the entirety of Shiffman's code, although the accompanying lines of commentary originally a part of the files have been removed:

```
HashMap words;
String[] tokens;
int counter;
PFont f;
void setup() {
  size(640, 360);
  words = new HashMap();
  String[] lines = loadStrings("dracula.txt");
  String allText = join(lines, " ");
  tokens = splitTokens(allText, " ,.?!:;[]-");
  f = createFont("Georgia", 36, true);
}
void draw() {
  background(255);
  fill(0);
  String s = tokens[counter];
  counter = (counter + 1) % tokens.length;
  if (words.containsKey(s)) {
    Word w = (Word) words.get(s);
    w.count();
  } else {
    Word w = new Word(s);
    words.put(s, w);
  }
  Iterator i = words.values().iterator();
  float x = 0;
  float y = height-10;
  while (i.hasNext()) {
    Word w = (Word) i.next();
    if (w.count > 3) {
      int fsize = constrain(w.count, 0, 100);
      textFont(f, fsize);
      text(w.word, x, y);
      x += textWidth(w.word + " ");
    }
    if (x > width) {
      x = 0;
      y -= 100;
      if (y < 0) {
        break;
      }
    }
  }
}
class Word {
  int count;
  String word;
  Word(String s) {
    word = s;
    count = 1;
  }
  void count() {
    count++;
  }
}
```

(Shiffman, 2011)

The code works by taking the entirety of *Dracula* and stripping out all punctuation so that only words, and the spaces between them, remain. Then, each word is checked against the current contents of the `HashMap` and, if it does not currently hold a position within the `HashMap` container, is added thereto with a value (`count`) of 1. If the word *does* already exist within the `HashMap`, then the `count` is increased by 1. Once this is completed, the check is repeated for the next word in the novel, and then the next, until the entire novel has been "read" in this fashion. In another discipline, this activity might be labeled "text mining" as a way to describe the act of extracting meaningful data from an otherwise opaque or unexamined source text.

As these checks occur, each word that has been computed is written onto the screen, with its frequency `count` value influencing the size of that word. As more words are checked, previously discovered words are pushed to the right of the screen, and then up it as they move beyond the horizontal bounds of the screen. Eventually, words that were first checked early on in the novel are pushed off of the screen; they do not stop being checked and having their frequency recalculated, but the expressed results are no longer visible. In other words, there is a distinction between what the human user and the machine experiences, with the former engaged in a temporal visualization of *Dracula* that is relatively absent from the code.

There are several meaningful activities that occur within the lines of the `HashMap` code that, together, dramatically alter the concept of reading and writing. Perhaps the most significant of these is the incorporation of object-oriented programming (OOP) principles, evident in the construction of the `Word` class. In OOP, distinct “objects” - clusters of expressive code entities—are created from a framework of rules (called “classes”); each instanced iteration of an object obeys the same procedural rules but responds independently to those rules—that is, it calculates and expresses its procedures without any inherent influence on any other object. The result is a set of objects whose behavior has the same fundamental principles but emerges uniquely for each individual based upon the constraints of its being called or created as part of a given program. (As an aside, OOP is unrelated to the recent emergence of the philosophical sub-field that has been named object-oriented ontology for its focus on non-human entities.) What OOP allows for is the potential for an *inherent* multiplicity of contextual meaning, made possible through the expression of iterative object creation and activity emerging from an initial set of algorithmic procedures.

At the same time, the nuanced context of each iteration of a given class is somewhat flattened; as the `Word` “Lucy” is counted by Shiffman's program, the meaning of each use of the name is stripped away. There is no remaining distinction between when Lucy speaks, when she is referred to by other characters, or what sort of emotional tenor surrounds those utterances. Instead, the name, as a `Word` defined by the code, becomes notable as a unique string of alphabetic characters that appears so many times over the course of the novel, represented visually by its expressed size on the screen in contrast to other word strings. Lucy—or any other word from the novel—gains significance through numerical frequency. Because of the way that Shiffman coded his program, there are a few intriguing side effects of this shift in significance (see Figure 2.1, below).

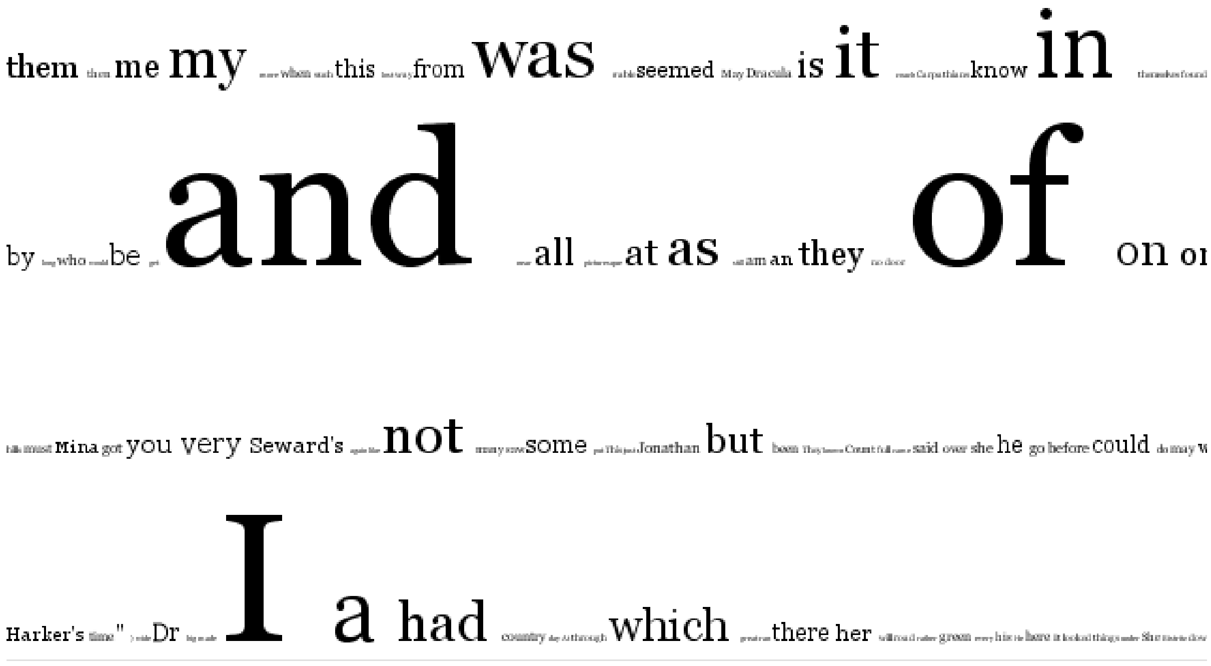

them me my this from was seemed May Dracula is it Carpathians know in
by who be and all at as am an they of on
Mina got you very Seward's not some Jonathan but Count said over she he go before could
Harker's time" Dr I a had country although which there her green here

Figure 2.1: Early expression of Shiffman's (2011) HashMap code

Early in the expression of Shiffman's code, it becomes clear that the most frequent words in *Dracula* (and, potentially, most novels) are words that might conventionally be considered the least important, the conjunctions, articles, and prepositions that link together more “important” concepts and messages. It is possible to add into the code exclusionary conditions that would ignore these seemingly trivial words; however, any subsequent analysis of word prominence via frequency would be inaccurate. Further, the decision to reveal the computations of word frequency in this manner reveals a very different visible outcome than in another manner, such as generating a text list of each word and its final frequency count. Because some words in Shiffman's program are pushed off the screen, the user ultimately is left unaware of the significance of words that appear early in the novel (see Figure 2.2, below).

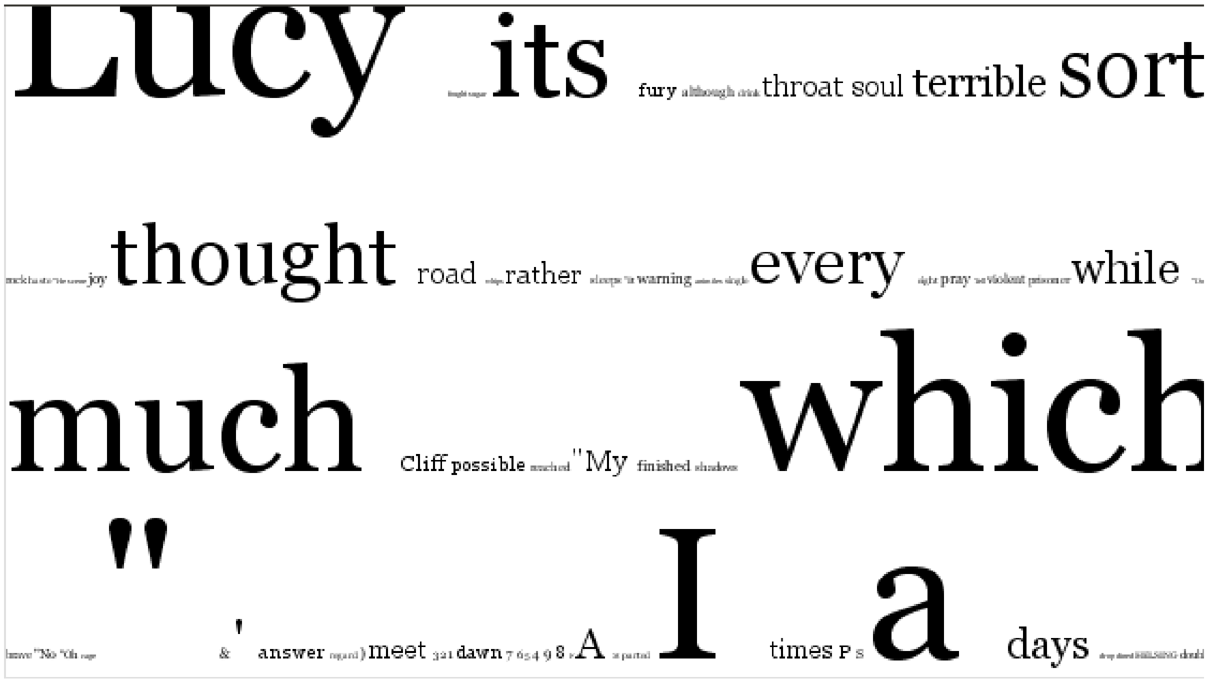

Lucy
its
fury
throat soul terrible sort
joy thought road rather warning every pray violent while
much Cliff possible "My finished which
"
&
'
answer meet dawn A I times a days

Figure 2.2: Later expression of Shiffman's (2011) HashMap code

In Figure 2.2, a screenshot taken twenty minutes after Figure 2.1, there is a noticeable difference in the makeup of visible words. While the code continues to compute every word that it scans in the novel text, the user is only given a small window into its results. The words “of,” “not,” and “Mina” have all been repositioned beyond the bounds of the program's output window, obscuring the visualizations of their frequency; meanwhile, “Lucy” has gained prominence near the top of the screen, hinting at the possible importance of other characters named earlier in the novel than she was.

In addition, there is an observable glitch in Shiffman's code: the characters `"` and `'` are not only recorded as iterations of `Word` but are clearly significant frequency-wise. This raises the question: why are these characters left out of the original line of code meant to strip away non-alphanumeric characters? The following line of code handles that function:

```
	tokens = splitTokens(allText, " ,.?!:;[]-");
```

The function `splitTokens()` takes any string—in this case, the content of *Dracula*—and separates out any chunk of text (i.e., words) from one another by using the included characters as delimiters. That is, the string “Hi, Bob!” would be recognized as possessing two “worthwhile” elements: “Hi” and “Bob” as the comma and exclamation mark would be flagged as points at which to break apart content on either side of their locations. However, the means by which to include quotation marks or an apostrophe within the set of delimiters would have its own glitchy consequences. For example, “Mina's” would no longer be distinct from “Mina” because it would be broken into “Mina” and “s” with the former adding to the `count` of “Mina” appearances that had not originally appeared in possessive form within the novel.

Why might something seemingly so trivial be worthy of discussion? The choices Shiffman made in creating this program signal to user and developer a particular set of values related to reading and (re)writing *Dracula* through algorithmic means. Word-level elements of the novel are deformed contextually from their original meaning and given new context through significance—the words that appear more often in the text appear more prominent in the code's expression. At the same time, the way that Shiffman has chosen to determine what “makes” a word (based on the characters that might surround it) alerts other coders, *and the system expressing the code*, as to how to arrange and read the text he inputs to be parsed.

This is not simple instruction devoid of persuasion: we are influenced to (more likely) accept a developer's meaningful text-making by the symbolic action engaged in by all participants of the algorithm's construction, interpretation, and computation. As part of this action, we may more effectively evaluate how successfully the author-developer and the computer have been in generating and demonstrating this argument for a new form of reading.

2.4: Conclusions

More generally, each of these examples discussed above serves to demonstrate some fundamental qualities of code as a significant form of rhetorical meaning-making. While on its own each program may seem to achieve only a limited purpose, together the code functions provided in these examples work—with thousands of others—in more robust software programs to persuade and influence numerous human and technological audiences to act in rhetorically meaningful ways.

Further, the potential for meaningful symbolic action that is demonstrated in these examples reflects the close connection that has existed between for millennia between algorithmic procedure and humanities activity, creatively and critically oriented work alike. While the vast majority of rhetorical focus has thus far centered on a variety of communicative modes including writing, speaking, image, and even place, the scholarship on *procedure*—and especially algorithmic procedure—as means of persuasion is relatively sparse. However, rhetorical code studies serves as a space in which to tease out the rhetorical potential of algorithms and of software code as a particularly significant contemporary form of constructing algorithms to facilitate action. To support this goal, the following chapter focuses on the spheres of discourse *surrounding* the composition of code; an examination of written communication about code can provide a way to help demonstrate the rhetorical strategies used by programmers to develop and promote software. The values that developers stress, in addition to the tactics they use, highlight a set of qualities likely to be communicated in and through the code they produce.

CHAPTER 3: RHETORICAL PRACTICES SURROUNDING CODE DEVELOPMENT

In the previous chapter, I focused on the longstanding relationship between algorithmic procedure and humanistic activity both before and after the advent of the personal computer. The argument I presented ultimately demonstrated how algorithms composed in code have persuasive potential and engage in rhetorical action for both human and technological audiences, albeit to differing degrees for each of those types of audience. In this chapter, I take a step back from a focus on code to explore the types of discourse that occur *around* code and its development, with a specific emphasis on the communicative practices shared by groups of programmers as they collaborate on software programs, often via email listservs, comments in code files, and in other asynchronous online forums.

This sort of discourse assumes a prominent role in this chapter specifically because it is clearly rhetorical, in that developers persuade one another to accept the choices each has made in developing particular chunks of code (which, as I observed briefly in the previous chapter, are themselves persuasive acts). Each developer often takes a position where he or she must defend the rhetorical choices made in composing a particular piece of the overall software, and the arguments made in that defense bring to light a number of values possessed by that developer about the program and its purpose(s); the language(s) upon which the program is built; the development team and its goals; and even the broader perspective that the developer may possess about what is possible in code, and how it can be made possible within the constraints of that particular project.

In the following pages, I emphasize the relationship that code-related discourse has with traditional forms of discourse studied by rhetorical scholars. The rhetorical qualities of these types of communication—including online email and forum discussion as well as networked software systems dedicated to tracking different versions of a code project—can and do tell us a great deal about the considerations developers weigh when it comes to discussing their work with collaborators. While this sort of discursive activity only partially describes the range of rhetorical appeal made by developers to development audiences, it nonetheless helps us set the stage for a discussion, in the next chapter, of rhetorical activity in code itself.

3.1: Rhetorical Scholarship Regarding Online Discourse Communities

Rhetorical inquiry into the makeup and activity of online discourse communities (formed by a wide variety of shared interests or qualities) has occurred primarily through two major approaches. The first approach primarily examines online communities as electronically transplanted iterations of conventional communities, with their forums and email listservs as sites for gathering and exchange that are perceived to function as digital *agorae*. The second approach explores online communities as fundamentally distinct from conventional communities; online communities are generally positioned as networks with unique qualities that would be impossible to replicate in non-digital environments. While these two approaches differ from one another significantly, they nonetheless are similar in regards to the interactions on which they focus: the traditional patterns of discursive communication between individuals within a given community. These approaches are complicated by an integration of code into the rhetorical examination of online communities and particularly those communities involved in the development of software, for whom code is not merely a facilitator of discourse but a means of communication in its own right.

Admittedly, many of the studies that make use of the first approach described above —viewing online communities as digital extensions or iterations of conventional communities—are relatively dated in nature, having taken place during the late 1990s or early 2000s during the early years of the internet's acceptance by the general population. In part, many of these studies likely focused on the conventional qualities of online communities due to the limitations or unanticipated uses of contemporary technology, such as Benson's (1996) examination of Usenet groups' political discourse or Silver's (2005) discussion of electronic shopping center as town hall. More fully, Gurak (1997) examined a variety of online community responses to two failed initiatives, Lotus Marketplace (a marketing database program for small businesses) and the Clipper chip (a proposed encryption device to be overseen by the US government). Gurak's studies focused on concerns of privacy possessed by involved parties and the various types of protest they staged; however, many details of these protests were discussed by Gurak through more conventional media with a broader audience and mainstream appeal, such as articles about the controversy appearing in the *Wall Street Journal* (pp. 23-24). When she turned to online postings about either of these controversies, Gurak's analyses of their qualities, although initially described as “profoundly different from anything yet experienced in communication technology” (1997, p. 44), ultimately framed online discussions as hybrids of conventional forms of speaking and writing. Specifically, internet postings were perceived as having both the immediacy of speech (in the general forum or marketplace) and the permanence of writing (utterances that persist through time and capable of being replicated exactly).

A number of scholars agreed in large part with Gurak, although most have been keen to qualify her argument as to the specific attributes of electronic communication. Barker (2000) suggested that online writing is primarily as described by Gurak above (both permanent and immediate) with the exception that it could continue to be edited or reworked even after being made “permanent” through publishing (i.e. the content and design of a web site). Warnick (2007) highlighted the potential problems of access, literacy, and clear author identity regarding digital technologies, each of which might hinder an otherwise more complete online reconstruction or iteration of more conventional public discourse. Hocks (2003), meanwhile, focused on the inherent multimodality of digital environments in order to emphasize the distinction between traditional rhetorical situations and the break from that tradition which she argued was taking place on the web.

More significant is the growing number of scholars who, in the last decade, have advocated the second approach to online community rhetoric as described above, in which digital environments and persuasive situations are considered fundamentally distinct from those surrounding conventional communities. Brooke (2009) called for a new media reconsideration of the rhetorical canons and of the trivium of classical education. His work has been the most notable and direct attempt to examine how new media demand new rhetorical perspectives in order to understand how we communicate and persuade one another in ways unanticipated in antiquity. Feenberg (2002) emphasized the influential role that technologies play in the formations of communities that make use of particular technologies, constraining and altering the decisions that might otherwise have been made by a community's members. Shirky (2009) observed the contributions of self-organizing communities to construct meaningful and dynamically shifting arguments through projects like Wikipedia. Other scholars, including Carpenter (2009) and Carnegie (2009), have focused on the interfaces of online environments as boundaries to (and facilitators of) forms of literacy that enable and constrain particular means of discursive interaction. In particular, Carnegie (2009) questioned the traditional qualities of electronic discourse by pointing out the new forms of *exordium* that emerge from new types of interfaces, many of which have simply been unavailable in other rhetorical situations.

The question that fuels this second approach is: what, if anything, makes online communities and their discourse *unique* or novel? There is not yet any consensus among rhetoricians in response to this question, since the camp that views online communities as extensions of conventional communities generally equates digital technologies as tools to facilitate conventional discourse. In turn, those who may not view online groups in this way, such as Moxley (2008), have framed their arguments around the goals that these electronically mediated communities work toward. For Moxley, the idea of the “datagogy”

(p. 182) as a descriptor for the democratization of the classroom would be difficult if not impossible to execute outside of the internet since it requires massive real-time crowdsourcing to vet and modify invention, style, arrangement, and delivery for a relevant argument made both to itself and to secondary audiences. In other words, the community's ability to engage in this sort of activity *at all* distinguishes it from a conventional approach to deliberation and persuasion.

Each of these qualities described above, those conventional or unconventional, is significant because it helps explain the potential range of interactions among, purposes of, and rhetorical means available to the members of various online communities. While these methods provide crucial insight into the ways that communities' members interact, there has been little scrutiny of how online communities' members make and share meaning with one another through the code that comprises a significant part of their discursive interactions. This is especially important for communities whose shared interest is the software that their members work on—these are groups who do not attempt to recreate conventional civic communities but instead construct social networks influenced by development processes. Accordingly, the meaningful communication their members share through and about code is often provided in unique ways that just as often do not approximate traditional forms of conversation or deliberation as in the ways that those forms of communication do reflect traditional forms of discourse. Here, I turn to a discussion of communities dedicated to software development and to the "meta-discourse" that surrounds their code—the types of communication that describe, deliberate upon, or propose compositions in code related to particular software projects.

3.2: The Rhetorical and Social Makeup of Software Development Communities

Software development communities, like any other broadly defined groups, are widely varied in terms of overlapping interest, level of relevant expertise or knowledge, or intensity of dedication or enthusiasm among its members. Unsurprisingly, the computer-oriented identity of such communities is likely to play a significant role in how those communities construct themselves and in how their members interact. Specifically, the uses of electronic technologies as spaces for collaboration, deliberation, and socialization all influence the perspectives held by community members regarding how to properly and appropriately interact with one another.

This significance of technology as influence on community makeup and activity can be seen in the variety of potential demographics for a given software development community. For example, any such group might exist because its members are all employees of the same corporation (e.g., Microsoft, Red Hat, Adobe) or because they belong to the same professional organization (e.g., World Wide Web Consortium, Open Source Initative, Association for Computing Machinery). It may exist because its members are all professional developers, amateur or hobbyist developers, a mix of both, or its member base may include those interested in *but not necessarily involved in* the development process. It may exist because its members work within the same code language(s) or on software applications that assist with a specific variety of purposes. In many cases, the existence of a given development community is fluid if not outright ad-hoc in nature. Further, its members are likely to be dispersed geographically across the world thanks to an increased access to high-speed telecommunications technologies and infrastructures.

This fluidity is especially significant for communities focused on the development of open source software (OSS), since the defining principle of OSS is that the source code of a given program is distributed alongside the compiled, executable software package so that other interested parties can—if they so choose—modify and redistribute that source code out again, and so on. Specifically, OSS enables the possibility for *any individual so motivated* to manipulate or alter a program as he or she sees fit because public access to its code has been provided by its developer(s). This opening blurs the boundary between the conventional labels of “producer” and “consumer” in a profound way, one that opens up the potential for a hybrid identity—what has been dubbed the *prosumer*—as both developer and user of a given software program. With the advent of the prosumer, anyone might view him- or herself as a developer, or at least someone who can create content, if not programs. Such a sense of access and capability is central to the success of many OSS projects: many find themselves able to count themselves as members of communities that they might have otherwise been excluded from (or considered themselves excluded from). For example, the members of the Red Hat Fedora operating system's community are referred to in numerous documents as “friends” of whom each contributes, in some way, to the operating system's success. Because of this shift in perception, the dynamics of those communities—and thus the ways the members in them communicate—increase dramatically in variety of members' purposes for participating therein, including whether they choose to do so through improving upon the program's code or discussing ways in which it might be improved.

As a result, what makes software development communities stand out from many other types of social groups is this dual focus on the potential to create meaningful software through code contributions and to deliberate about how “best” (often individually qualified) to improve upon existing efforts to create that meaningful software. In effect, members of development communities can attempt to persuade each other in two distinct ways: first, in what they code (i.e., how the code communicates its intended purposes and abilities), and, second, in what they explain to one another about what they perceive their code to do say and do or what they perceive their code to *have the potential* to do. This is not to suggest that all community members will engage in all possible forms of persuasion but instead to observe how persuasive activity could occur; it is likely that those who may not feel comfortable exploring code may also not engage in its discussion explicitly, relying instead on secondary points of focus in order to stay involved. These points may include concerns about the user experience of a program or about the social culture that may surround its use.

3.2.1: Open source software development communities.

For the better part of twenty years, OSS has gained ground in the software market as an alternative model to the traditional system of proprietary production and distribution. The defining qualities of OSS include free access (meaning unlimited or unhindered, but often financially “free” as well) to source code and communal development and improvement of software programs. Individuals possess the right and, for some, the knowledge to experiment with and make adjustments to OSS programs. In addition, users are often encouraged to distribute the modified code back out to the general populace so that it can be further modified and distributed, theoretically *ad infinitum*. The most notable OSS projects are the Linux operating system (of which dozens of distinct distributions exist, including Red Hat's Fedora, Debian, Slackware, Ubuntu, Mandriva, and Gentoo), the Mozilla Firefox web browser, and the Apache web server, which has been said to be the world's most popular web server software. In fact, results from a Netcraft (2012) survey show Apache to hold 57.23% of the market share for web servers of all domains from August 1995 to November 2012. While these few programs listed here are easily the most well-known OSS projects, they nonetheless demonstrate a widespread use of OSS programs; this suggests in turn a growing popular interest in OSS development.

Despite a significant increase in OSS use and development since the early 1990s, it would be misleading to claim that OSS is *radically* different in terms of development structure than its traditional, proprietary counterpart. That is, in both paradigms, there generally exists a community of dedicated developers working on an active software program and a community of non-developer users that may or may not contribute suggestions for improvement (such as through bug reports of requests for features). Where OSS differs from proprietary development is that there often exists some overlap between “developer” and “user”—hence the increasing popularity of the term “prosumer” to acknowledge this overlap

—both in the amount of contributed aid provided by developers or users and the amount of perceived importance possessed by individuals in either group.

In a significant number of OSS networks, a hierarchy of development contribution value has been established wherein certain developers possess significant influence over a given project and the vast majority of end-users hold very little (Christley & Madey, 2007). A number of studies have attempted to understand the social dynamics of OSS development communities and any hierarchical structures constructed by those communities (Crowston et al., 2005; Crowston & Howison, 2005; Howison, Inoue, & Crowston, 2006; Wiggins, Howison, & Crowston, 2008). However, these studies generally suggest that a clear organizational structure, rather than a dynamic and chaotic environment, has been most often demonstrated as the most effective means of ensuring project success. In fact, Crowston & Howison (2006) suggested that the optimal structure of an OSS development community "is onion-shaped, with distinct roles for developers, leaders, and users" (p. 114). This onion has at its center the primary source of social power and influence, comprised of the project founder(s) and the release maintainer(s). Each consecutively outer layer becomes less influential and actively involved in the direction and improvement of the project, with the outermost layer consisting of those passive users who do not engage in contributing at all. Crowston & Howison's onion metaphor is helpful in describing the general hierarchies constructed for a particular software project, but it obscures the somewhat dynamic movement that can occur by individuals between these layers and privileges (or responsibilities). For example, an individual overseeing a significant section of a major OSS project might be near the central layer of the onion of that section, but he or she may be less central in regards to the development of another major section of the project, to say nothing of the project as a whole.

3.2.1.1: Rhetorical influences on OSS development social structures.

If an OSS community can be conceived of as an onion, how might we envision the hierarchical layers thereof? This is a difficult act, given the fluid nature of most OSS groups; Crowston & Howison (2006) identified one structure—the onion—but it applies to a particular moment in time and is approached from a specific perspective (i.e., the overall development of a project). Unfortunately, the onion metaphor fails to take into account the continuous rhetorical activity that occurs among developers as they work to influence one another on the specific contributions they offer to part or all of the community.

That said, the onion metaphor is nonetheless incredibly helpful in emphasizing the strata of valued roles and contributions to a project through a particular lens. Lakoff & Johnson (1980) have noted the integral role that metaphors play in how we think and communicate; by associating the structure of an OSS community with the layers of an onion, Crowston & Howison (2006) suggest both that there is an inherent structure (inner > outer) *and* that all the layers are, fundamentally, the same. This similarity is significant for OSS projects: even those who may not feel as though they are offering any substantial contribution to a particular program—or who may not be perceived to contribute in a substantive manner—are nonetheless often provided with the message that any and all contributions are worthwhile. Problems with collaboration and its seeming egalitarian nature have been identified by scholars for the last several decades, even as they ultimately stress the benefit of such work (McNenny & Roen, 1992).

The potential for rhetorically powerful communication among the members of a given community is greatly affected by the structure of that community. The conceptual organization of an OSS community as an onion suggests that there are rising levels of social significance and celebrity within increasingly inner layers, as they are both more involved in and valued by that community. In this sense, the community appears to function not unlike a more traditional discourse community or social entity—such as a central figure (e.g. the president) making large-scale pronouncements that affect the entire community, while individuals in less powerful positions appeal to community leaders or their intermediaries. However—as has been noted by numerous rhetoricians from Gurak (1997) to Moxley (2008)—electronic technologies have flattened the *perception* of hierarchy among a community's members. For example, since email can be sent to anyone else (or an entire organization) with an email address, one's apparent ability to persuade directly has been radically increased: anyone can voice his or her concerns and desires to the population at large. While the effectiveness of such a communicative method may not in reality be improved in any measurable way (that is, there is not necessarily a demonstration that other developers pay such a message any mind), it nonetheless provides a community's members with the *potential* to engage the entire community. With the need to maintain enthusiasm among developers (or possible developers) in order to keep a volunteer-maintained project active, this ability is central for the project's success.

Among the OSS development communities that have been notably affected by the unique qualities of hierarchical “flattening” upon discursive practices is Red Hat's Fedora Project, whose website described the Fedora Linux OS on its front page as a program built by a community that stresses a close relationship among its members: “Fedora is a fast, stable, and powerful operating system for everyday use *built by a worldwide community of friends*” (Fedora, 2012, emphasis added). Even though the vast majority of users may be unlikely to engage in direct development of the operating system, and even though the vast majority of those may never be considered for any sort of social leadership position, there is nonetheless an explicit appeal to *friendship* made on behalf of the community. By centering on friendship as the key descriptor of its social makeup, Fedora is able to position all of its contributors as laterally-situated developers rather than as members of a vertical hierarchy. But is this claim —or others like it—supported by practice? That is, do contributors to an OSS project actually possess this advertised bond with their colleagues?

It is incredibly difficult to determine objectively how social standing in any given OSS development community is generated or weighed. In many cases, developmental contributions are solicited from popular (user) audiences as a means of generating interest in continued improvement upon given programs, with the implicit message that contributing *anything* not only affords one entry into a valuable community but also affords one status therein. For example, the front page for Red Hat's Fedora distribution of Linux advertises several qualities that make the operating system great. Among them is: “Our community. People world-wide work together in Fedora to advance free software” (Fedora, 2012). It is not just that people use it, or that they collaborate in the system, but that they do so *in order to* improve and advance the project and others like it. Further, this activity is recognized and endorsed by the overseers of the project itself, since it is advertised as one of Fedora's core strengths.

Of course, it is unlikely that any OSS project would attempt to articulate how it rates or otherwise evaluates its members in any regard other than the merit of each individual's contributions to the project. For example, the development community for GIMP, the GNU Image Manipulation Program, has provided on its website a list of different ways that interested parties could involve themselves in contributing to the program, including "program[ming] new features[, …] writ[ing] tutorials[, … and] help[ing] others to learn to use GIMP" (GIMP, 2012). Not all of the proposed contributions demand the same skills or time of its developers. By casting a wide net over its community in terms of solicited contribution activities, the GIMP community is able to engage users that might otherwise have never considered themselves capable of involving themselves in GIMP's development process (such as those without programming knowledge).

However, this is not to suggest that all contributions are viewed equally by all developers in all communities. Given the hierarchical nature of most projects, as described by Crowston & Howiston (2006), it is much more likely that particular contributions—and particular *types* of contributions—are valued more highly than others. Often, those who possess or who are *perceived* to possess the most code-related experience on a project are likely to be in positions to influence the largest populations of community members. Linus

Torvalds, the inventor of Linux, is a particularly prominent example of this type: he has retained an semi-official role as “benevolent dictator” of Linux kernel development even though he is not the most prolific or dedicated programmer associated with the project (Ingo, 2006). In fact, in one recent interview, Torvalds noted that he no longer reads code but instead merely evaluates the proposed changes that his collaborators bring to him based on their *own* evaluations of the code submitted by others (Moody, 2012). In other projects, leadership roles may be passed from one individual to another in appointed succession or by selection of the “upper management” of a development effort. Still others, like the Debian distribution of Linux, elect an overall “project leader” through the use of a popular vote in which all interested developers can participate (Debian, 2011). In the case of Debian, the nominees highlighted the work that they felt best suited them for the position of project leader, and each nominee's perception of significant or valued work was substantially different from that of the others. As a result, at each election the community is provided with an opportunity to consider the direction of the Debian project in ways that might otherwise not be considered or discussed at length, if at all, by the general community.

While there are many communities that have similar general structures and philosophies, each development community is unique in its social makeup, even if many of its members share membership in several similar communities. In short, the specific goals that drive a particular project are not uniformly possessed by, or valued in, another. Stewart (2005) observed that social status in OSS development communities was driven in large part by communicative efforts as much as by any potential objective evaluation of merit. He noted that available social references informed the determination of specific developers' social standings (as opposed to a purely empirical evaluation of programming skill). Those interested parties who established themselves quickly in a community—as willing and able to accomplish desired development tasks—were more likely to be afforded a higher social standing than those who contributed at a slower pace after their initial interactions with that community (Stewart, 2005). As a result, *kairos* becomes an integral component of social capital as an individual's capacity to contribute significantly and meaningfully to a particular community: those who recognize, and make use of, opportunities in which to position themselves as valuable are often those who ultimately gain some sense of authority over the community's central project.

According to Nakakoji, Yamada, & Giaccardi (2005), the group-oriented goals of each development community are the adhesives that maintain connections between members and other projects. It is difficult to determine how long a given project's developers will remain active; as Abdullah et al. (2009) noted, the nature of OSS development relies on voluntary collaboration with little incentive beyond individual desires to improve the project under development. Howison, Inoue, & Crowston (2006) stated that while a given project may not change much over time at its center, i.e. in what the purpose of that project may be, the range of participants may be quite varied. Individual developers that are not directly involved in leading the community generally only contribute for a brief period of time before losing interest or motivation.

This motivational force is in line with the concepts of both *kairos* and rhetorical agency: there is a temporal and situational energy possessed by parties involved in a particular rhetorical situation that facilitates successful engagement with other involved parties (Miller, 2007). Such agency-as-engagement extends to the meaningful and rhetorical action of developing a particular program or a feature thereof. In many cases, persuasive motivation is successful specifically because it imitates, to varying degrees, the energy of a productive business environment; Ballentine (2009) has discussed the complicated nature of opening or obscuring source code in order to compete in particular types of markets and how those decisions affect subsequent work in those markets. That is, developers may contribute to a project as much to package and distribute the relevant product for industrial ends (e.g. profit, market dominance) as they do to collaborate or to promote OSS principles. This is due in many cases to OSS volunteers working on specific projects because those projects are valuable to the volunteers' employers.

Despite the noncommercial ethos of open source, it is nonetheless a powerful tool for corporate and proprietary purposes; O'Reilly (2005) has noted that many services built on OSS frameworks, such as Amazon and Google, are not required to share the applications they have developed because those services are simply *used* by consumers as black boxes rather than *distributed* to them as standalone programs; the agency of OSS freedom is co-opted into closed source, proprietary agency denied to interested parties not involved with those corporations. In other words, the rhetorical power generated by OSS communities through the OSS collaboration and distribution model is in many circumstances also sapped by proprietary competitors' ability to outmaneuver it for specific (business) ends. For OSS communities to function most effectively in achieving success, the rhetorical awareness of their members to engage one another in productive and collaborative communication is necessary. But how do, or can, they do so? And what do they communicate with one another to promote their goals? These considerations are addressed in the following pages.

3.2.2: Goals of OSS developers.

The goals of most OSS communities are widely shared, even if they are not always explicitly noted: very generally put, each project that developers collaborate on is, hopefully, advertised and distributed to increasing numbers of users who might decide to contribute to the project's continued development. However, this broad goal does not properly articulate the range of purposes held by portions of a development community (individual and group alike). For some, there is a desire to make use of a program for a particular productive end, such as for the benefit of one's regular employment. For others, there is a desire to direct the development of a program in a certain direction, such as to conform with a community's “best practices” or to facilitate one's own vision. For others still, there is an effort to demonstrate one's skills as a programmer in order to bolster one's reputation or to find employment. Others may simply want to experiment with code or particular functionalities and share their work with their colleagues.

The most commonly shared goal, as noted above, is the increased distribution and use of a particular OSS program. This goal is not necessarily a *universal* goal; there may be users who are uninterested in promoting it to a broader community. However, given the success of most OSS projects as reliant upon a regular stream of new contributors to replace those who become inactive, it is not surprising that “survival” becomes a driving factor. In almost all cases, the appeal to survival and prosperity is intertwined with developers' attention to kairos. For example, despite initial fanfare, the Diaspora and Identi.ca microblogging and social network platforms have thus far failed to upset Twitter and Facebook as prominent forms of digital communication despite the free and comparatively open approaches taken to their services and methods of software distribution (Terdiman, 2008; Lawson, 2011). In comparison, the Mozilla Firefox browser—about which more detail is provided in the following chapter—has prospered in large part because it emerged as the successor to an already popular and influential browser (Netscape Navigator) and because it was perceived as a preferable alternative to the *de facto* browser packaged with Microsoft Windows (i.e., Internet Explorer, or IE). Hundreds of developers flocked to the Mozilla project because of a discontentment with IE. For some, it was the lack of cross-platform support for IE (meaning that only Windows users could use the browser). For others it was a desire to promote open, rather than proprietary, web standards so that web development would not risk being locked into IE-related development.

3.2.2.1: Standardizing practices.

Most OSS developer goals, however, are more nuanced and less ubiquitous. A large number of developers are interested in streamlining code practices in line with individual project style guidelines. This sort of effort improves code readability across files as it helps readers learn to expect particular approaches to problem-solving and the like. However, it also potentially inhibits other developers' contributions if they do not follow those guidelines. Alternatively, these standardizing efforts may *rework* contributions to effectively remove the individual character of its author from the file, making each OSS contribution a balancing act between the preferences and needs of the community and the preferences and abilities of the individual. The developer's ability to persuade the larger community that a given contribution is helpful “as is” impacts how significantly different any proposed changes may be offered in response to the initial contribution and, in turn, how different the resulting code ultimately becomes when and if it is merged into production.

Concerns of standardization and normalization are key to a critical and rhetorical understanding of algorithmic procedure as meaningful communication. That is, the processes by which algorithms are made to conform to particular operational paradigms (often referred to as “standardization” or “optimization”) for computational efficiency as well as for developer readability can tell us a great deal about the values possessed by a given community in regards to its product(s) and the functions thereof. This is not to suggest that there are no circumstances in which specific operations or functional patterns *are* the best means of achieving particular goals but instead to observe the persuasive influence of the computational and rhetorical *methods* by which developers pursue those goals and by which they deliberate their work. Just as Bogost (2007) described the procedural rhetoric of computational systems teaching users how to operate within them, so too can we consider here the rhetoric surrounding the construction of procedures in order to facilitate those procedures' activities.

One such discussion took place among França et al. (2012) in regards to a proposed change to Rails, a popular OSS web application framework. Rails works as a service that makes use of a very readable programming language called Ruby to deliver content to users. Rails builds on a three-part structure of models, controllers, and views that serve to describe different rhetorical functions of the data being manipulated: briefly, models describe data objects in a general or ideal (Platonic) sense; views render them into the output accessed by users; controllers deliver data requests to the proper destination so that views can be rendered properly. Examples of well-known websites that are powered by Rails include Twitter, Groupon, and Hulu. While there are hundreds of contributors to Rails, there exists a core team of established developers who collectively decide on the future of the project, and França is a member of that team. França's proposal added a symbol, a special type of object in the Ruby language on which Rails runs, to a small portion of the Rails code. The overwhelming response to this proposed change was that the symbol was unnecessary; as one contributor phrased it, “I think this is a horrible addition to the router that doesn't appear to be actually solving a problem, as well as poor code” (França et al., 2012). França's contribution butted up against the set of practices that had been established and supported by the majority of vocal contributors to Rails, and he appeared to be unsuccessful in convincing his colleagues that it was a worthwhile addition to the project.

However, David Hansson—the initial developer of Rails and a “benevolent dictator” over the project with the ability to overrule even the core team—ultimately decided to accept the proposed code, noting, “I find [the proposal] to be a wonderful addition to the domain language and a key building block for making beautiful Rails applications. So we will spread that beauty far and wide” (França et al., 2012). In essence, Hansson accepted the proposal not because the community decided it was the best approach but instead because *he* felt that it adhered to his own preferred style for Rails code. The remaining discussion hinged on whether Hansson's decision was a wise one, focused on whether his own preferences for how Rails code is written and how it works may no longer be in line with the stylistic preferences and goals of the majority of Rails developers. Even though França initially appeared to be unsuccessful in his contributory attempt, his goals in constructing the code *as he did* were aligned closely enough to Hansson's goals (and Hansson's personal views on the ends of the Rails community) to result in an acceptance of França's proposal.

3.2.2.2: Self-promotion and experimentation.

A number of individuals involve themselves in OSS development either to hone or show off their coding skills, putting their contributions to the community in order to learn from, or have their work validated by, their peers. This is not to suggest that individuals with such goals are necessarily working *against* the general aims of the larger community, but instead to observe that one's effort to persuade others to accept one's code contributions may be fueled by more than one purpose.

For example, there are several contributions currently awaiting review for (and, theoretically, eventual merging into) the central production of the Mozilla Firefox browser. The details of the review process for Firefox is elaborated upon in the next chapter due in part to its hierarchically complex nature, but in brief: Firefox is maintained across several software versioning systems and makes use of multiple programming languages (C++ and Javascript), each of which has its own established set(s) of development standards and practices. Of the outstanding reviews for Firefox, one is particularly noteworthy due to its author's request that it be neither merged nor denied since, he argued, it was "only for review purposes. There is no intention of landing this change" (Gozalishvili et al., 2012). In other words, Gozalishvili utilized the collaborative nature of peer review upon his code primarily as a way to *improve* his code abilities by bringing his efforts to the attention of his fellow developers. The comments provided by other users followed through in this regard, focusing on the understandable and readable nature (or, rather, the lack thereof) of the submitted code (Gozalishvili et al., 2012). One commenter suggested that Gozalishvili improve the readability of his code by providing more illustrative examples to explain his intent; Gozalishvili responded by suggesting the commenter "just read docs [documentation], putting too much examples [sic] is not that useful" (2012). The commenter replied again after more code submissions:

> So, inch by inch, this is getting closer to the quality required for mozilla-central. Even if we omit our core disagreement on how to implement `result`, [sic] there is still

> work to do. Unfortunately, by now, I am just too tired to continue pulling you kicking and screaming to that level of quality. (Gozalishvili et al., 2012)

In essence, the development community almost unanimously stressed a particular route through which the author could alter his work in regards to their shared project, offering feedback that could extend to code efforts beyond the scope of the Firefox browser. At no time did the community suggest that Gozalishvili *stop* contributing but instead deliberated on how he might best accommodate the larger population of involved developers by improving upon his code and communication skills. While Gozalishvili often pushed back against these suggestions, the rhetorical considerations of his contributions were more and more acutely weighed until he turned to revise his code thoroughly. Even though the specific proposals under consideration have yet to be fully approved or denied, Gozalishvili's awareness of how his code works *outside of* its mechanical operations—that is, how other coders view and value it—has only improved, in turn increasing his chances of being valued more generally as a programmer regardless of his place within the Firefox community.

Gozalishvili's approach to explicit learning-through-doing can be understood relatively easily through the lens of composition studies, in which collaborative and multimodal work has been a major focus for research. Emig (1977) argued that writing was a unique form of learning, and while she distinguished it from other forms of symbolic composition (such as mathematical equations), we can consider her conclusions in light of code. The writing process provides immediately accessible results, but code demands its execution for its potential to be realized. Code, like writing, is often “epigenetic, with the complex evolutionary development of thought steadily and graphically visible and available throughout the record of the journey" (Emig, 1977, p. 127). This is especially true for versioning systems like GitHub, which allowed Gozalishvili to learn not just from others' code but from his own as it was revised multiple times in response to feedback from other developers.

The beneficial results of collaboration have been explored by a number of scholars, especially in regards to writing in digital environments. Moxley (2008) observed the positive effects of crowdsourcing as a way of creating and evaluating public knowledge: the "wisdom of crowds" can influence the instructional capacity and engagement of a community far beyond the ability of an individual to do so, thanks to the communal sharing and evaluating of information (p. 183). While the ethos of every individual member may not ever be fully confirmed or investigated, the crowd as a mass entity likely can and will norm itself to ensure that the most accurate and helpful information is disseminated to its members. It is important to note that there is a programming maxim, often referred to as "Linus' Law" (after Linus Torvalds) which shares this sentiment: "given enough eyeballs, all bugs are shallow" (Raymond, 2000). In other words, the number of individuals scrutinizing and editing a particular text—whether conventional or code—is proportional to the text's overall strength and accuracy as it is revised by its editors.

3.2.2.3: Evaluation of contributions.

While communities like Firefox leverage their hierarchical structures into educational opportunities for aspiring developers, other development communities have chosen to focus, or at least emphasize, their efforts on establishing an equal playing field—or at least the perception thereof—for the projects on which their members work. Others stress the meritocratic, and thus beneficial, makeup of their community structures. Such efforts promote a more inclusive and inviting atmosphere for contributors that might otherwise be hesitant or resistant to participating, since there can often appear to be a substantial barrier (in-depth knowledge of code) to entry into a particular development community. This sort of appeal is especially intriguing in that it does not necessarily reflect the actual dynamics of a group but attempts to persuade individuals that advertised egalitarianism is the norm.

The Apache Software Foundation (ASF) is the developer organization formed to support the development of the Apache web server, and its official website documentation highlights the meritocratic nature of Apache-related development. The ASF positions itself not only as a meritocratic community that recognizes the value of its contributors, but as *an exception* to the social character of other software development groups. On a web page outlining how the ASF works, the following described the foundation's self-perception of its meritocratic leaning: “[U]nlike in other situations where power is a scarce and conservative resource, in the apache group newcomers were seen as volunteers that wanted to help, rather than people that wanted to steal a position” (Apache Software Foundation, 2012). Simply put, the ASF claims to have prospered because its take on social “power” is that it is a non-precious “resource,” meaning that theoretically any and all could join in to help. However, this freedom *is* restricted, as noted in that same document: while the ASF “was happy to have new people coming in and help, they were only filtering the people that they believed committed enough for the task and matched the human attitudes required to work well with others, especially in disagreement" (Apache Software Foundation, 2012). Only a select few were granted permission to alter the production code directly, those that "had 'earned' the merit to be part of the development community" (Apache Software Foundation, 2012). In other words, the ASF self-policed by supporting those individuals its members felt were "committed enough for the task" as opposed to those who might contribute in a more casual or occasional fashion—and only those in the former category were considered to be *part of the community*. It was thus not enough that individuals might want to help but that they demonstrated a desire and ability to focus on the contributions they made to Apache, whether through frequency, amount, or quality. The meritocratic ASF community is actually a set of communities, including those who are viewed as having earned a prominent place and those who have not yet earned such a place.

These examples demonstrate in a small way how particular individuals and groups attempt to position rhetorically the work they do, and the ends they pursue, for various audiences. The goals and their discussions provided above will be compared below with developers' rhetorically-aware appeals to determine how frequently persuasion is explicitly employed to achieve a given end versus appeals that implicitly suggest code as the sole focus of deliberation.

3.3: Developers' Rhetorical Awareness of Their Coding Practices

While it is certainly possible to discuss developers' rhetorical practices regardless of their explicit awareness of those practices, we can gain an even greater understanding of how they choose to communicate with one another by looking at the extent to which they *do* recognize the rhetorical qualities of the appeals they make to one another. We can begin by recognizing that developers' discourse—and the code texts they create—as rhetorically informed and significant forms of meaning-making for specific purposes. While rhetoricians would rightly argue that any decisions made as part of development activities are fundamentally rhetorical in nature, examining rhetorical self-awareness here is important as it allows us to scrutinize the *intentional* practices of invention in which developers participate in order to engage in the composition of software programs.

In some cases, direct connections between rhetoric and code practices are explicitly drawn by developers. Ford (2005), has identified rhetorical goals at the heart of his work, using the idea of coding *elegance* to help drive the simplest and most flexible way to achieve one's goals. Ford specifically compared the development philosophy behind the Processing language with that of the web, and he asked web developers to consider the principles behind their work: "Is the browser the right way to navigate the Web? […] Why are some semantic constructs more privileged than others? […] How can content truly be reused?" (pp. 84-85). Ford (2005) also recognized the rhetorical qualities of interfaces and code alike; he called implicitly upon Lanham's (1993) bi-stable oscillation of text (simply put, looking *at* a visually-appealing artifact and looking *through* it for some symbolic meaning) as a way of understanding more effectively the history of web development (or any software development) to date (Ford, 2005, p. 91).

Ford's final question on rhetorical oscillation is integral to code practices in that it emphasizes the persuasive nature of effective code and communication: “What is *sprezzatura* for the Web?” (2005, p. 91). Sprezzatura refers to one's ability to make a difficult act *appear* as though its execution were effortless. Ford's question is significant since it focuses on rhetorical style and delivery: asked in other words, how can the effective and successful construction of code make web use easy and accessible for all? In essence, the rhetorical value of user and coder goals should influence development practices (how can the members of either group communicate effectively without such efforts resembling a chore?) rather than be considered only in retrospect as a quality not inherent in software creation or use.

In a critique of general software practices, Platt (2007), argued that many code decisions stem from considerations of convenience for a developer rather than from some objectively superior or “best” quality connected to those decisions. In other words, many developers lack the rhetorical awareness required for a successful engagement with one's user base or audience. As part of a description of the 'Do you want to save the changes?' dialog box that appears when a user attempts to exit Microsoft Notepad without saving the file he or she has been working on, Platt (2007) observed that

> [t]he programmer wrote the program this way (copy the document from disk to memory, make changes on the memory copy, and write it back to disk) because that was easier for her […] Reading or writing characters from the disk (spinning iron

> platters with moveable parts) is roughly a thousand times slower than doing it in memory (electrons moving at the speed of light)[.] (p. 17)

Despite this increased speed and machine efficiency, Platt's argument suggests that the program's functionality works *poorly* here because any successful use of Notepad demands an understanding of how the programmer intended the program to work. That is, the dialog box's question only makes sense if the user recognizes that files are not saved to disk by default: the programmer is “forcing [the user] to understand that she's written the program this way” (2007, p. 17). It is important to note that, while the approach provided by Platt is faster than the alternative, the decision to use *either* is fundamentally rhetorical in that the workload-related interests of the programmer—reduced code-based program preparation via defaulting to memory (RAM) storage rather than disk storage—ultimately overpower the interests of the user, which might very well include automatic saving of data to the hard disk.

While developers may not always, or even often, draw attention to persuasive strategies that do not focus solely or primarily on code, *when they do*, these strategies can provide rhetoricians with tremendous insight into the ends to which those developers work and how they attempt to influence their fellow developers about embracing those ends. It is rare that a programmer will make explicit reference to the language of rhetoric (e.g., almost no one will refer to *ethos* or *chiasmus*) but there is a good chance that he or she will mention his or her credibility (such as the amount of experience that individual has with the project or the code languages used on it) or personal emotional investment in regards to a particular issue. When developers lean on the fundamental rhetorical appeals in these ways, the ends to which they persuasively work, and the relationships they perceive themselves having with other developers or their projects, become visible in intriguing ways. In addition, these persuasive activities can enlighten us as to the ends to which particular code texts and paradigms are constructed and promoted.

3.3.1: Rhetorical appeals used in community discussions.

Developers' use of rhetorical appeals in order to convince developer audiences to act is relatively varied between individuals. Just as a rhetor relies on the fundamental appeals of ethos, logos, and pathos in order to persuade an audience in a more conventional setting, so too do developers turn to these strategies when it comes to discussing the merits of particular approaches to code. I suggest there is no *fundamental* difference between using the appeals as part of either a code-based or a more discursive effort at persuasion, although the types of constraints for each mode of communication may sometimes be distinguished easily from those of the other. Perhaps unsurprisingly, many programmers ultimately argue for code practices that hinge on concerns *outside* of the code itself—such as when well-known developers stress their social position as support for their claims. Developers' recognition of their rhetorical efforts is important here, since the *ways* that they construct and present their arguments is as insightful about their values on coding as the code texts on which they deliberate.

3.3.1.1: Ethos.

Among the appeals used relatively explicitly in development communities is that of ethos, which is used not only to demonstrate the credibility or expertise of a particular individual but also to "legitimize" the project to (or *from*) which a particular set of contributions are made. Several rhetoricians have explored the distinctions between traditional appeals to ethos and complicated contemporary appeals to ethos that are less clear-cut. Miller (2001) has suggested that ethos is central to electronic communication because humans need to be able to evaluate their discursive companions: "what sort of character is behind the words: one we can trust? one we can learn from? one who is like us or one who is strange and challenging? one we can dominate or one who will seek to dominate us?" (p. 273). These concerns, while not always easily answered, nonetheless fuel much of an audience's determinations of a particular rhetorical effort. Warnick (2004) argued that web communication, precisely *because* it is often so difficult to discern the author of a text (let alone his or her character), demands a consideration of ethos focused more on the performance of credibility through the presentation of information than the expertise of a clearly-defined author.

These concerns over the relationship between conventional forms of ethos and newer means of demonstrating ethos are debated in many communities. For example, Nutter et al. (2012) discussed code proposals for a number of security tests related to the code suite distributed with the Ruby programming language. This proposed code had initially been developed for a fork of the language that ran inside the Java Virtual Machine and thus could work alongside the Java language (and this fork was called JRuby due to its hybrid nature). The JRuby community's members felt it had the potential to benefit the broader Ruby development community and offered to merge it into the main Ruby project—as long as its origin in JRuby remained clear: they “would like to contribute these tests, but [JRuby's developers] would ideally not lose the JRuby bug numbers for future reference” (Nutter et al., 2012). Since the code was still “in process,” the bugs that JRuby developers might continue to find could make their fixing more easily implemented. At the same time, there is an implication that Ruby developers might seek out JRuby if they, separately, composed workarounds for those same bugs.

Nutter's offer is thus contingent upon the community's willingness to recognize the authority of the JRuby community for its contributions. Not surprisingly, this appeal was met with some resistance by Ruby developers; one individual who noted his appreciation for the code asked in response to the request for JRuby's bug references to remain in the code, “I'm confused, I was thinking I could simply strip those references when committing it to [the main Ruby project], or not?” (Scott & Bosslet, 2012). Here, the ethos of the JRuby community clashes with the ethos of the code itself (as worthy to be included in Ruby), bringing to the spotlight the differences between the values of each community. Regardless of the proposed code's ability to solve problems that might be faced by both the Ruby and the JRuby communities, the clear line of origination back to JRuby (and its own developmental processes) to be included alongside that code remains contested as of the time of this writing.

A counter-example to the JRuby case above can be found in a discussion related to how scalable vector graphics (SVG) image files are implemented through jQuery, a popular Javascript library that enables event handling and animations on web pages. One developer noted that it was impossible to hide SVG images in the Mozilla Firefox browser and offered a potential fix to that problem; subsequent discussion focused on whether it was—or should be—the goal of the jQuery community to resolve problems that they identified as belonging to other communities. As one respondent noted, “this seems like a lot of work to support something we don't support,” while another likened the proposed solution to a can of worms (Sherov et al., 2012). Almost unanimously, the development community determined that work related to this issue was *not* within the purview of their collaboration. The only real point of disagreement reflected the lack of clarity as to who *should* take care of this issue: is it Firefox's domain? Or that of other Javascript libraries? There has no formal consensus to answer these questions, in part because no individual has assumed a position of authority in order definitively mark out that territory. Nonetheless, the community's demonstration of their shared recognition of their boundaries (i.e., that they exist) is significant. Here, the developer's ethos is as much about understanding the *limits* of individual and group authority and expertise as much as understanding when to play up those qualities.

3.3.1.2: Pathos.

Perhaps unsurprisingly, developers who are dedicated to the projects they work on often find themselves balancing their discussions between 1) technical and logical reasoning and 2) passion for the work they contribute to their projects. While it is rare to see a discussion in which emotional appeals serve as the *primary* means of promoting particular perspectives, pathos nonetheless remains a critically important strategy through which individual developers can stress the strength or intensity of their convictions—or at least the convictions they perceive their audiences to feel strongly about—in relation to a particular point or to support appeals to ethos. Fahnestock (2011) has observed this relationship between ethos and pathos as they relate to stylistics: "[a]ttitudes and bids for alignment are encoded in every language choice, and the rhetor's presence and relation with an audience are the unerasable ground of all discourse" (p. 279). For discussions around code, these language choices are especially important since they are meant to reflect the emotional response not just to an individual comment but to *the code under discussion* as well.

One particularly noteworthy conversation revolved around a request for code to be merged into the general Linux kernel through the code-sharing website GitHub, which makes use of a software versioning system, a means of tracking software changes across a distributed population, named git. This request required approval from Linus Torvalds, the "benevolent dictator" of Linux development; what made the request *notable* is that Torvalds adamantly refused to address any code administrated through the GitHub site. He argued:

> Git comes with a nice pull-request generation module, but github instead decided to replace it with their own totally inferior version. As a result, I consider github useless for these kinds of things. It's fine for *hosting*, [sic] but the pull requests and the online commit editing, are just pure garbage. (Zhigunov et al., 2012).

According to the hierarchy of the Linux development community, Torvalds' decision was appropriate and authoritative; his stressing that GitHub was "pure garbage" served to demonstrate the reasoning that led to the decision. However, as many others observed, the proposed code change was *three lines* long—a trivial amount of code to absorb into the project. As a result, Torvalds' argument stands out due to the intensity of his conviction against GitHub as a system for software administration despite the absurdity of his immediate rejection of such a brief amount of code.

Torvalds responded to these criticisms of his stance further in the thread, and his claims were supported by the continued metaphor of *garbage* as a description of how GitHub works. Specifically, he noted that

> the reason for that is that the way the github web interface work [sic], those commits are invariably pure crap. Commits done on github invariably have totally unreadable descriptions, because the github commit making thing doesn't do *any* [sic] of the simplest things that the kernel people expect from a commit message[.] (Zhigunov et al., 2012)

While Torvalds expounded upon the reasons for which he felt GitHub failed as a version control system, his equating the website with "garbage" or "pure crap" remains the most compelling appeal for his position. The emotional tenor of Torvalds' argument was further exposed in another of his comments, responding to a developer who disagreed with his stance: "The fact that I have higher standards then makes [sic] people like you make snarky comments, thinking that you are cool. You're a moron" (Zhigunov et al., 2012). Further discussion on GitHub's implementation of git's code merge request capabilities revolved around the stylistic preferences held by each participant in the conversation. Few comments were dedicated to logical argument; most centered on the “objectively best” practice that would benefit not only Linux development but all software development hosted by GitHub.

Ultimately, Zhigunov's proposed code was never implemented through his GitHub submission, but Torvalds' pathetic arguments clarified for many developers how continued work on the Linux kernel would progress from that point forward. As many recognized, the rational *merit* of Torvalds' argument was not any more important or central to the project than Torvalds' personal opinion. As Garsten (2006) has noted, emotion, and especially passion, can “stimulate reflection or judgment by disrupting ordinary habits of response” (p. 196). The Linux kernel development community was obligated to accept Torvalds' complaints as a normal component of community discussion in order to continue “work as normal” through the GitHub versioning system. However, since his complaints directly addressed that system, an inordinate amount of discussion focused on whether or not Torvalds' criticism was accurate rather than whether the code proposal merited a merge into the main code repository. As Torvalds had the final say in how the community's development practices were structured, his position was essentially impervious to criticism—leaving him free to offer arguments that allowed him to share the emotions that fueled his perspective.

Just as there is no universally agreed-upon “best practice” for coding in a particular language or even on a specific project, neither is there a consensus on the most appropriate means of presenting an argument for a particular community. The kairotic qualities of a coding situation influence each developer (rhetor) and audience in unique ways, and the appeals used by a developer to persuade his or her audience highlight that developer's understanding of the relationship he or she has with the project under discussion.

3.3.2: Non-traditional rhetorical activities surrounding code.

While online forums and email listservs remain the most popular, discursive, and accessible forms of communication among developers, they are hardly the only means of meaningful engagement that do not also involve writing in code. These activities, which both make use of code texts *and* non-code discussion, offer substantial insight into the rhetorical concerns that drive software development with particular goals in mind. In some cases, the goal is to reinvigorate a project by splitting its community (and the authority that guides it); in others, the goal is to merge together individual experiments with, or to improve upon, the established product. No matter the goal or the means by which a developer takes action, there is a fundamentally rhetorical component that influences both the developer and his or her audience to participate in further development and meaningful communication about the code being composed for a given project.

3.3.2.1: Forking.

The concept of *forking* contributes significantly to the ebb and flow of many OSS community life cycles, and it does so in fundamentally rhetorical ways. A fork is a cloned version of a project that has become distinct from its originator project at a particular point in time, and there may or may not be code or developers shared between original and forked projects from that point forward. Well-known forks (and their originators) include Ubuntu, a fork of Debian Linux; LibreOffice, a fork of the OpenOffice suite; and WebKit, a fork of the KHTML web browser framework (WebKit powers the Google Chrome and Apple Safari web browsers). A fork may be created for one of any number of reasons, such as an individual desiring to tinker with a program outside of the project's regular work flow or development trajectory. In other situations, the fork may have been created because of a philosophical schism between several developers that made further collaboration impossible. Despite this relatively dark example, Howison (2006) has argued that in most cases, even when projects may have regular forking occur, there are often efforts by the forking developers to provide improvements for the original program as well as for their forked versions. This is likely because continued inter-communal good will increases the chances of other developers contributing back to the fork in turn as an implicit payment in kind.

Despite its technological focus, forking bears a close resemblance to the back-and-forth of conversation. As ideas are passed back and forth within a discourse community, they are tested, supported, refuted, and mutated as different individuals choose to address them. Sometimes, requests for responses are clearly indicated (such as when a developer comments in code on “hacking” a workable but non-preferred solution to a problem, unable to come up with a stronger case until a colleague suggests one); at other times, responses are provided but not solicited once a program's code is “published” alongside the program itself. Burke (1973) provided what remains one of the greatest descriptions of discursive activity, comparing knowledge creation as deliberation in a social setting:

> Imagine that you enter a parlor. You come late. When you arrive, others have long preceded you, and they are engaged in a heated discussion, a discussion too heated for them to pause and tell you exactly what it is about. In fact, the discussion had already begun long before any of them got there, so that no one present is qualified to retrace for you all the steps that had gone before. You listen for a while, until you decide that you have caught the tenor of the argument; then you put in your oar. Someone answers; you answer him; another comes to your defense; another aligns himself against you, to either the embarrassment or gratification of your opponent, depending upon the quality of your ally's assistance. However, the discussion is interminable. The hour grows late, you must depart. And you do depart, with the discussion still vigorously in progress. (pp. 110-111)

For Burke and for many rhetoricians since, this metaphor of the parlor reflects academic and civic discourse: our understanding of a topic is advanced only through its public "testing." Contributors to the discussion—individual scholars or citizens—add their voices when they can, but the discussion continues long after each departs. The same is true of software development: forking serves as a means of "testing" a set of ideas by allowing motivated individuals to test alternatives to an original development plan. The strength of a particular argument (development philosophy) then ultimately is determined by the success of a particular fork. While it is the project's development philosophy that propels its creation, a fork might prosper as much from the kairotic solicitation of aid and use (that is, establishing a solid user and developer base) as from the strength of the program's code base.

The practice of open source software distribution and access—of which forking is a crucial component—could itself be viewed as a form of rhetorical imitation: at what point does the use of another developer's code blur into a subsequent developer's work? The absorption of another's work as a foundation for one's own, in a programming sense, often has more in common with rhetorical *topoi* than with academic plagiarism (although this is also certainly within the scope of discussion). As Miller (2000) observed, "[t]he *topos* is conceptual space without fully specified or specifiable contents; it is a region of productive uncertainty [… I]t is a space, or a located perspective, *from* [sic] which one searches" (p. 141). *Topoi* are useful rhetorically to assist a rhetor in constructing a particular argument by revealing and generating paths for that rhetor's expression from the potential lines of reasoning he or she could potentially pursue. In regards to software developers, for a particular method or function to be used effectively or to be perceived as being used effectively—that is, to accomplish a specific task—it is expected to be constructed and used in a certain manner, following the general style and conventions of a specific development community, so that other developers can build off of that construction. Forking, which makes use of a preexisting form of a given project, thus reflects the ability of a rhetor to make use of a particular *topos* for his or her immediate and situated purposes.

In some cases, a forked version of a project may ultimately be *merged* back into its originator. As with forking, this adjustment in a project's existence reflects a dynamic shift in the social makeup of the development community. A fork may have been intended to exist temporarily, such as to experiment with a single feature whose capabilities may not have been fully tested as part of the “production” version of a program. A fork may have been created with the intent for it to be separated entirely from its originator as a distinct project in perpetuity. However, just as arguments and ideas are sometimes folded back into certain lines of inquiry and discourse, so too are software forks often absorbed back into the familiar development structure and community from which they had initially split. This potential for merging is just as significant as the potential for forking: if an OSS development community truly is dynamic and constantly changing, then a split (fork) cannot ever be considered permanent—just as the makeup of that community cannot ever be considered to be stable. Forking is also noteworthy in that it functions as a dynamic counter to the static metaphor of community-as-onion described earlier in this chapter. When forking is considered as a potential vector for any individual developer looking to adjust his or her position within (or relationship to) a given community, the possibilities for social and rhetorical readjustment are expanded. Even though many forked projects' developers contribute to their originator projects, there is no obligation to do so. The tenor of the development conversation changes when a given community gains or loses focus in regards to a given perspective; given that this shift in focus is *constantly* occurring, a consideration of development as a trajectory of forking and merging paths is appropriate.

3.3.2.2: Pushing and pulling.

Many software versioning systems, like git (which powers GitHub), make use of a two-pronged form of code sharing called *pushing* and *pulling* (described below) that together resemble nothing so much as deliberation in regards to the social construction of knowledge and action. The dichotomy between pushing and pulling is much like that of persuasion in general, which Garsten (2006) has described as

> prone to two forms of corruption. […] In our desire to change [the minds of an audience] lies the danger of manipulating, and in the effort to attend to their existing opinions lies the risk of pandering. The two vices thus arise from the dual character of persuasion itself, which consists partly in ruling and partly in following. (p. 2)

In other words, a successful attempt at persuasion occurs when there is a well-navigated path balanced between overt manipulation or pandering. Miller (2010a) directly compared this dichotomy with what she called the "push-pull model of technological development" in which innovation draws audiences, while elsewhere the lack of innovation spurs others in new directions (p. ix). For collaborative software projects, this comparison is especially valuable. However, the means by which a balance (between manipulation and pandering) is perceived to be attained may vary wildly between communities. The potential interactions that may take place through pushing and pulling simultaneously draw attention to, and obscure, rhetorical negotiations on persuading developer audiences to accept and improve upon particular code-related decisions.

Pushing works for code distribution in a versioning environment much like a classical oration: it is disseminated from an authoritative figure (such as a core development team for a particular software program) to any and all interested parties, like those who may potentially become involved developers. As new versions of a program are released, whether with major or minor improvements or revisions, the development team pushes the code to a tracked repository from which any other developer may *pull* that code for their own use. In other words, pushing works as a kind of one-way broadcast transmission of a message that is both computational and rhetorical. Just as it provides updates to executable code, so too does it suggest that the decisions made to construct that code reflect the goals and values of the development team. Implicitly, there is, accompanying the code itself, an argument that any code which other developers want to have included in future releases of that program should work and look *like the code being distributed*—an achievement complicated by the inherently dynamic nature of continually-developing code practices.

In contrast, *pulling* serves as a relatively passive acceptance of a rhetorical effort, an audience-oriented evaluation of the code being pushed out to its broad user population. Garsten (2006) referred to this sort of approach (in a general sense) as "pandering" in that it demands of the rhetor-developer an approach which plays down the rhetor's contributions in favor of the audience's expectations. That is, individual users, at least for OSS projects, have the ability to modify and redistribute (or fork) freely the code for a given program. What they usually lack is the size of the audience of the original or "mainstream" version of the program, which means that their work on the project may likely be overlooked. Many software versioning systems provide a workaround for this, allowing users to make *pull requests* to the core development team: a pull request effectively works as a plea to merge together the individual user's code changes with the mainstream code base (as demonstrated in some of the examples provided earlier). This process provides a voice to the individual by giving him or her the opportunity to demonstrate the rhetorical and computational power of a proposed code change. At the same time, when it is merged, the identity of its author is somewhat erased: the code content becomes a part of the main program and, barring any specifically-added comments to identify its creator, the code appears identical to the other lines of code surrounding, or other files accompanying, it. Given the hierarchical structures that comprise many OSS communities, the result is that a developer may need to construct his or her code in such a way as to appeal not only to the needs of the user base but to the stylistic and logical preferences of the project's authorities. The pull request serves to highlight the "vanishing act" that occurs through successful rhetorical appeal: the user (and his or her work) is normalized and assimilated into the project's body of work and activity.

Ignoring pushed updates is a possibility as well, and this action promotes a unique rhetorical approach to communication and collaboration. As discussed earlier in regards to forking, the idea of "breaking off" from an established community works directly against the political and rhetorical dynamics of a "top-down" model by providing an autonomous identity to the motivated developer—at least until a new community emerges around the forked program. After that, there may be a new (or reoccurring) model of hierarchical structure that influences subsequent work in regards to that fork, and the "pushmi-pullyu dynamic" of rhetorical and technological appeals is reactivated, "leading us to engage in or to attempt certain kinds of rhetorical actions" (Miller, 2010a, p. x). This potential for a reinvention of existing social and rhetorical systems raises an interesting question for rhetoricians: how does forking (and its subsequent further forking via voluntary updates or voluntary ignoring of updates) promote engagement within a rhetorical ecology? It seems both to emphasize the audience's ability to choose its own individual voice(s) and to eschew the discursive potential of an established project by iterating deliberation through fragmented productions of differentiated programs. These varied forks are unlikely, in most circumstances, to have their code interwoven into others' forks after establishing themselves —and this ultimately provides an intriguing, and possibly unique, model of rhetorical appeal through the avoidance of direct conversation.

3.3.2.3: Patching.

Other than forking and collaborative versioning, perhaps the most "social" practices of software development is *patching*, where small fixes to specific code issues are applied to existing software programs. Patches serve as a sort of continued discourse between parties (developer and client) as security bugs are fixed, hardware functionality is updated, and so on; often, the exigence for a patch stems from explicit communication from a user of the software, requesting improvement in some regard. Patches are often highly *kairotic* in that they are developed quickly by individual programmers who spot exploits on their own installation of a given program and then perhaps distributed out to the broader community for that program. Platt (2007) highlighted one traditional weakness of this practice: patches "work only if you use them, and almost nobody does of his own accord" (p. 85). That is, software patches are generally only applied when a particular client notes some problem with his or her program and seeks out a potential fix to that problem. In recent years, this issue has been mitigated somewhat by automated updates that in many cases are partially or fully "invisible" to the end-user: Microsoft Updates and Ubuntu Update Manager, for example, each regularly check for patches and then request users to update relevant programs, either individually or as a “batch.” Ubuntu's Software Center and Apple's Software Update feature work similarly.

While this increased automation makes implementing official patches easier, there is an issue of control, not unlike the issue with Microsoft Notepad addressed by Platt above. In one sense, code concerns become a non-issue as users are directed to merely apply the regularly-provided updates from developers. In another sense, though, code concerns become a *central* issue, as there is no demand for users to examine proposed updates on their own and determine whether or not to apply them. While there may be users who do approach automated updates with caution, the system assumes users will simply follow the suggestions of the update management software. With manually-applied patches, there is an expectation that a user recognizes a particular issue and is actively seeking a resolution to it. There are two significant possibilities arising from this approach. First, the user may simply not recognize other issues to fix and leaves his or her software open to other vulnerabilities (and, in a sense, all software is open to unacknowledged vulnerabilities). Second, the user specifically avoids patches that he or she feels are unnecessary to apply. This second possibility is important in that the user, while not unilaterally protecting his or her software to the same degree, is capable of controlling the “bloat” of the software by only increasing its size and processor use with the additional functionalities and protections that are desired. In addition, it must be noted that the security of automatic updating hides the potential *new* problems that might be caused by patch code: just as with revision of any other type of writing, even though one issue might be fixed by a patch, the added code might in turn generate new issues that need to be addressed by *more* patches, and so on.

Accordingly, not all patches or updates are equal. In some cases, they are designed to fix seemingly trivial but potentially devastating issues. For example, Voswinkel (2012) fixed the problem of a typo relating to an account-specific variable in the code for microblogging platform rstat.us. This typo left open the possibility of "orphaned" code, or a set of references to data that no longer existed. Specifically, the Ruby variable `:dependent` had been initially written as `:dependant`, which meant that any other functions attempting to make use of the variable would not locate it properly anywhere in the software. Even though this particular issue was extremely easy to resolve, the participating developers—the "fixer" as well as the developer who had initially coded in the problem—identified their roles clearly for public scrutiny since discussions *about* that update were recorded alongside the updated code itself.

Often, patches cause any number of unforeseen and significant consequences for users and their systems, making the "fix" the catalyst for problems that would never have been encountered if the program had *not* been updated. For this reason, patching generally involves the possibility for a user to back out of the update—making software versioning (the use of differently-updated iterations of a program) a unique phenomenon among meaningful forms of communication. While any individuals participating in some discourse might revisit previously-stated ideas and reconsider their positions, there is no "undoing" of the reconsidered discourse; for code, however, this can literally be the case, and new issues can be erased outright by a rollback to a previous version. As a result, users who have opted to employ various versions of a given program often find themselves engaging in otherwise identical activities that, by necessity (e.g. being unable to patch successfully) or desire (e.g. avoiding problems likely to result from a patch), reflect specifically situated contexts stemming from the capabilities of their currently-used software versions.

The Drupal content management system, for example, is widely used and considered highly successful because of its numerous customizable “plug-and-play” modules, software add-ins developed by volunteers to help website administrators achieve specific tasks. Each module has its own set of dependencies and effects on the overall Drupal system, meaning that some modules work in unexpected fashions when other modules are also installed, and many developers recognize the potential for catastrophe here. The sheer number of potential module combinations that one could install makes any sort of anticipation for particular bundles of modules almost impossible. As a result, a common practice among Drupal developers is to provide patches to problems they identify and then ask volunteers to point out any problems stemming from specific modules present in their installations.

Such a practice treats the development of a given module (as well as of Drupal at large) as a continually emerging process of fragmented innovation and standardization, as developers are tasked both with experimenting in new ways with technological capabilities and ensuring that the program is as likely *not* to fail or break when used by the largest population of administrators and users as possible. When the software does break, then, the moves undertaken by involved developers present clear indicators of the rhetorical awareness of the current situation (fixing an unforeseen problem) and of the purposes various administrators might have for setting up particular module combinations in their Drupal installations.

For example, one very commonly-used module is drush (short for "Drupal shell"), a command-line tool that allows Drupal administrators to modify their systems quickly from a terminal window rather than through the default browser-based interface. When used in combination with git, a program that provides versioning control for collaboratively-developed software, drush—or almost any Drupal module—can be updated and tweaked regularly and relatively quickly. However, differences in software versions of both drush and git between individual Drupal developers can quickly lead to complex and complicated situations in which it is difficult for all involved to easily smooth out problems experienced by some or all. When a developer attempts to change "working versions" of a program to test out changes to code, git should track the version currently under development. For Drupal users jgraham et al. (2009), however, the ways that git and drush (specifically, a sub-command of drush called "drush make") worked at the time disrupted the successful progress of development for either program, and the ensuing discussion provided a representative example of the rhetorical and discursive nature of composing and revising code. jgraham noted that a perceived similarity in command syntax for two git command flags (`--working-copy` and `--bare`) should have, but did not, provide similar results for a git-related update, as both flags define the location of a "working" directory in which development would occur, although the `--bare` flag specifically changes the hierarchical directory structure of the code project. However, functional differences in multiple versions of git and drush used by developers resulted in failed patching in multiple ways: some users received error messages, while others received successful reports, only to discover the code they *thought* had been patched had not been.

To resolve these issues, jgraham et al. (2009) created several small patches that attempted to unify calls for one flag (`--working-copy`) to use the other flag (`--bare`) so that all potential developers would have the same project structure on their systems. However, problems persisted: those who reported using git versions 1.7.0.x found the issue existent, while those using newer versions of git perceived it as resolved. At the same time, drush as a project was being updated by a larger body of coders, and the command being problematically patched (drush make) was being merged as a project into a "core" project for drush—meaning that further development progress might vary wildly in focus and scope towards the problem jgraham had identified. Even though specific problems had been identified, some of which were even addressed and resolved, the rhetorical concerns of the ad hoc development community were not fully satisfied: the code-related constraints of individual Drupal administrators, and their proficiency with Drupal, drush, and git (or their desire or ability to upgrade any or all of those components) further limited each in achieving complete success in regards to this technical issue.

Such practices for patching and resolving, successfully and unsuccessfully, problems with software are not limited to Drupal; whenever developers collaborate on projects with diverse goals and on varied computer systems, contextual issues arise—in a fashion parallel to the contextual difficulties of rhetorical communication in any other form. When the members of massive volunteer communities of developers collaborate on software projects, such as in the case of Mozilla Firefox, the rhetorical influences developers and computer systems exert upon one another are myriad in significance and purpose; in some cases, the intended *action* says more about the various goals for the program held by individual or specific groups of developers more than the community at large.

3.4: Conclusions

Just as with explicit forms of discussion, so too do collaborative activities in code, such as forking and patching (among others), demonstrate rhetorical character and value as fundamental components of meaning-making through these activities. The potential for rhetorical interaction between developers and technologies through dynamic OSS community structures and development processes is especially important when considering activities like forking, patching, and naming. The meaning constructed and communicated through these practices suggests a set of tools by which developers actively engage one another with important and (potentially) effective rhetorical strategies that are employed *in the writing of code* as well in the writing that takes place around code. While at times these rhetorical efforts may be difficult to recognize—since they may not clearly reflect more conventional forms of discourse—they nonetheless work to persuade readers (e.g., collaborators) to engage in particular types of action or activity as much as to articulate the operations to be executed by the computer.

In the next chapter, these rhetorical strategies and others in (and around) code will be explored in more depth through the lens of software development undertaken on the Mozilla Firefox browser, a popular internet browser program collaborated on by thousands of developers over the last fourteen years. By examining a selection of code texts and practices composed by the Firefox development community, I will demonstrate how the browser's code (in the form of this set of examples) has rhetorically impacted subsequent development, *as well as use*, of the program for particular ends. Where persuasion occurred in relatively conventional text and activity in the examples provided earlier in this chapter, the forms of engagement undertaken through code will not always closely resemble traditional discourse. However, they nonetheless create and communicate meaning in rhetorical fashions—and it is important for scholars interested in code as a means of communication to recognize how and why this occurs.

CHAPTER 4: EXPLORING CODE FROM A RHETORICALLY CRITICAL PERSPECTIVE: THE CASE OF MOZILLA FIREFOX

In the previous chapter, I examined the rhetorical potential regarding several key forms of discourse that often surround and describe code practices and texts. In this chapter, I will build on that inquiry in order to clarify a rhetorical understanding of code. To do so, I engage in an in-depth analysis of code in (and leading to) meaningful communication as rhetorical action. There are numerous levels of code, including programming languages and systems of meaning as communicated through interfaces, and it is rare that a software program will not make use of multiple levels in order to function as the developer(s) and users desire. Any of these languages or infrastructural systems could, and *should*, be examined to illuminate this avenue of scholarship. For the purposes of this project a single code artifact—the Mozilla Firefox browser—will serve as the specific object of study, whose code is composed primarily in two high-level programming languages (C++ and Javascript). Part of the reason Firefox was chosen was because these languages reflect syntactically and grammatically natural languages like English, making the texts discussed below more accessible than if the code were abstracted further away from more accessible language.

As a rhetorical artifact, Firefox is comprised of a software program's source code, its compiled “executable” files, the conventional discourse surrounding developmental coding efforts, and the trajectory of its development processes over a set period of time. While these latter two qualities were discussed in depth in the previous chapter, that analysis could not be complete without an examination of code *in addition to* commentary. Thus, each of the means of persuasion listed above offers a unique lens into the rhetorical potential of code as object of study, especially as meaningful practice. In addition, each lens helps demonstrate how code-related discourse engages varying audiences and co-rhetors to initiate, participate in, or otherwise facilitate certain types of action.

In the following pages, I turn to focus exclusively on code as rhetorically significant and powerful forms of *text* as well as of practice. Of special interest is the range of action enabled by and through code-based persuasion, meaning the ways that audiences are influenced and persuaded by communication in, rather than about, code. This sort of communication affects activity not only explicitly *in or of* code's making (such as continued development and code composition) but also to the sort of user-centered interactive software engagement that occurs when the program is executed. The relative accessibility of the languages making up Firefox with will provide significant assistance in helping the rhetorically-minded reader to recognize how persuasive efforts within the code operate for various social ends.

To perform this analysis, I will examine several selections of code from the Mozilla Firefox web browser that offer insight into the range of rhetorical appeals that its developers have made through the code they have contributed to the program, primarily through the high-level, object-oriented programming languages C++ and Javascript. Firefox has been an open source project for over a decade, and thousands of programmers have been involved in its development. As a result, the sorts of persuasive efforts made in the browser's code can tell us a great deal about how members of the development community have effectively communicated meaningfully with one another about their preferred means of structuring the browser so as to enable particular types of activities.

Mozilla Firefox is one of the largest and longest-running open source projects in existence, with over a decade of open history and development to which thousands of volunteers have contributed. Firefox is developed by a globally distributed network of volunteers engaging in regular, publicly-archived discussion about improving its code and who, in some cases, have done so for well over a decade. The browser relies on operating system (OS)-independent web protocols to function, although there are differences supplied in OS-specific versions of the program so that it can be used successfully in one system versus another (such as Windows 7 as compared to Mac OS X). In addition, Firefox's publicly-available source code makes it capable of being modified by any interested and motivated individual. We can see in the browser's code the collection of efforts to promote rhetorical action in a number of interconnected venues, from stylistic nuances explicit in lines of code to discursive appeals made in developers' mailing lists *about* lines of code and how "best" to structure them. As outlined in the previous chapter, these appeals may not always overtly take into account rhetorical concerns and strategies, but they are nonetheless present *and crucial* to the collaborative development of the program. Inside the code for Firefox, we can often trace how particular rhetorical decisions impacted subsequent work on, as well as discussion about, the browser.

It is important to note that Firefox, like many software programs constructed from the coding efforts of numerous developers, is fueled in large part by the energy of corporate and institutional motivation and writing practices. By this I mean that, while its code is fundamentally rhetorical, that code is unlikely to appear *explicitly* creative or rhetorical in many of the examples included in this chapter. One can consider the difference between a mundane conversation and a State of the Union address: both are filled with efforts to make meaning and to persuade an audience, but the latter text is generally described as more rhetorically powerful or more likely to be performed by a skilled orator. Programming as meaningful work like programming is often defined in regards to the strictest understanding of its productive purpose, such as Nardi's (1995) objective-based explanation: "[t]he objective of programming is to create an application that serves some function for the user" (p. 6). Despite this relatively limited view of what code is and what it is for, there have been insightful developments in several fields (e.g. rhetoric, technical communication, media studies) that point to the potential for an understanding of code practices and texts as rhetorically valuable.

The closest such movement may be the study of the rhetoric of science, since the rhetorical decisions made by scientists in order to persuade scientific or public audiences often hinge on persuasion through tools and explanations that may seem—at least initially—*arhetorical* in content, style, and delivery (Fuller, 1997). However, as Ceccarelli (2001) observed, "Some of the best research in the rhetoric of science undertakes the close reading of individual scientific texts to show exactly *how* they were designed to persuade scientific audiences at particular moments in history to acknowledge the truth of their authors' theories” (p. 3). I mean to suggest that a similar, although not entirely parallel, line of inquiry is available through the study of code texts, including those whose syntax and purpose may be relatively distant from the conventions and makeup of natural discourse.

At the same time, I want to stress that much of the rhetorical importance of a given code text, as with other forms of text, is as much in the sort of action and activity it induces as in the specifics of the text's content. As Muckelbauer (2008) has argued, the very nature of persuasion "is not primarily interested in *what the proposition is* [of a given argument ...] Instead, it emphasizes *what the proposition does*, the responses it provokes and the effects it engenders" (p. 18). In order to understand code as a type of persuasive communication, we need to recognize this potential for action. Further, we need to recognize *how* an argument proposes action, since that consideration informs and influences how an audience will engage the proposition. Code suggests through its logical structures and operations particular ways of engaging with certain ecologies comprised of technological and human agents alike (e.g., computer processor, developer, end user). How user experiences are anticipated and facilitated, how data is calculated and transmitted, and *how a program should be further developed* are all types of engagement undertaken through persuasive arguments in code. Keith (1997) described rhetoric—when considered as an architectonic or “design” art—as “concerned with design of language for purposes both individual and social” (p. 234). Where Keith has mentioned *language*, we should extend that thought beyond conventional definitions of discourse—what many of us commonly equate with language—to include all manner of symbolic action, especially (for rhetorical code studies) algorithmic procedure and the logic of software code.

Admittedly, there is some difficulty in describing the rhetorical practices of code with the vocabulary of conventional rhetoric. Prelli (1989) outlined the concerns necessary for analyzing the ethos of scientific rhetors, given their frequent appeals to skeptical or disinterested engagement with their arguments (as discourse and as the seemingly-objective presentation of data). Gaonkar (1997) has suggested that for such efforts in the rhetoric of science, any attempt to clarify "the dialectic between implicit and explicit rhetoric[,] makes the very idea of rhetoric undecidable [...] Any critical text can be shown to possess a level of reflexivity that makes it rhetorical. The lesson is invariably that there is no exit from rhetoric" (p. 75). That said, Gaonkar's argument does not suggest the analysis of "implicit[ly]" rhetorical texts is less worthwhile than that of conventional texts—instead, he suggests that all meaningful communication is rhetorical, and it is the attempt to distinguish between *types* of conscious or unconscious influences on rhetorical suasion that is misguided.

An examination of the assorted rhetorical discursive and code-based efforts described briefly above can help us begin to answer significant questions that surround and further define the goal of rhetoric in an age increasingly influenced by digital technologies. Such questions include, but are not limited to, the following:

1. What sorts of rhetorical appeals are constructed and communicated directly within the various layers of code, in terms of logic and how that logic is constructed so as to be expressed effectively?

2. What sorts of appeals are offered in intra-code discourse—specifically in non-computational comment lines of text "explaining" code functions above, below, or next to those comments?
3. Further, what implications might we recognize as emerging from the influences each type of appeal exerts upon the forms of communication that extend across levels of interface and interaction?

Admittedly, critical efforts to understand how answers to these questions emerge out of specific, situated instances of rhetorical activity may not be clinically repeatable across multiple situations in ways that more empirical methodologies might be able to recreate. However, inquiries into these concerns can nonetheless provide critical understanding into the complexities of persuasion in and through code, and the contextual influences upon specific communicative activities will make each set of texts and practices unique. Further, the insights gained by examining code as a rhetorical form of communication have the potential to enlighten us as to rhetorical communication, and the means of persuasion available to a rhetor, in a broader sense.

4.1: Mozilla Firefox: A Code Study

In addition to having an incredibly large and active development community, Firefox is one of the most popular web browsers currently available (behind Google Chrome and Microsoft Internet Explorer), with an estimated 21% share of the browser "market" as of November 2012 (W3Counter, 2012). Firefox is also the browser with the greatest amount of development that takes place primarily through community-based volunteer efforts utilizing an open source software philosophy; while its major competitors are as “free” in a financial sense as Firefox, their source code is not freely available to the same degree. By existing as an open source project, Firefox's code and all of the discourse surrounding its development is publicly accessible, allowing interested parties to explore the trajectories of development, collaboration, and conflict over a decade of activity. As a result, Firefox serves as a rich object of, and site for, inquiry into the converging and interacting influences of code and conventional discourse on rhetorical action taking place as part of and response to the development of the browser.

4.1.1: A brief history of Mozilla.

The Firefox web browser was originally conceived in the early 1990s by employees of the Netscape Communication Corporation as a program called Navigator, based on one of the very first internet browsers, Mosaic—in fact, “Mozilla” as a name comes from a combination of Mosaic and Godzilla, the latter name referring to the Navigator project as “a beast vastly more powerful” than the former (Hamerly, Paquin, & Walton, 1999). While Navigator was offered to the public at no charge, it suffered in terms of gaining popularity thanks to increasing efforts by Microsoft to integrate its own web browser—Internet Explorer—into its Windows 95 and subsequent versions of the OS. After several years of diminishing browser share, Netscape released the source code for its newest version of the program under an open source software license and turned over primary development responsibilities to the global community in the form of the Mozilla Project (Mozilla, n.d.b). The Mozilla Project, in turn, assumed a new form in 2003 as the non-profit Mozilla Foundation, collecting together under its governance the assorted open source projects contributing to the Mozilla software suite (Mozilla, n.d.a).

The shift from corporate to open source and volunteer-based development marked a major shift for Mozilla's web browser project. The browser's code would be publicly and freely available, meaning that anyone interested could not only download the browser and its code but that person could also modify the code as he or she saw fit; further, one could release his or her own modified version of the program, assuming its source code was *also* made freely available alongside the compiled, executable version of the browser. Communication (i.e. email) related to the program's development would be publicly available, meaning that any interested parties could engage in discussion about the project and involve themselves in code *and any other* communicative efforts to influence Mozilla's developmental trajectory. The semi-organized, semi-chaotic nature of the developer community for the Firefox browser provides rhetoricians with the capability to examine how these wildly different types of social interaction are weighed against the other for rhetorical, influential purposes.

With Mozilla's projects existing as a set of connected but distinct development communities working on related software programs, the potential was initially overwhelming for rhetorical chaos to affect the progress of *any* of these programs, to say nothing of the suite as a whole. However, with almost fourteen years of collaboration and tradition to “normalize” the broad behavior of the development communities within Mozilla's fold, contemporary rhetorical concerns for Firefox's developers are quite different in many ways than they had been—although the voluntary and open nature of the projects means that some of the initial chaos remains a fundamental component of developer socialization and interaction.

Examining the development of Firefox through its code practices and texts is important not only for the field of rhetoric and for technical communication but also for scholars interested critically in the cultural dimensions of software and code. Firefox and the events and texts described in this chapter provide a lens through which we might more fully understand the social and industrial influences exerted upon software programming paradigms as well as how we might approach more clearly and expertly the user experiences anticipated by developers through the code structures they implemented in their programs. Where rhetoric focuses on not just what is said in a given argument but how it is said and what it induces, inquiry into the culture of development might explore (as an example) how social structures and conventions impact the compositions and interactions of particular communities and the sociopolitical ends to which they might work. These sorts of concerns are incorporated into the discussion below, although they serve primarily as threads to be tugged on by other scholars interested in the influence of culture on code development.

4.1.2: The turn to open source software: Ramifications on Firefox's development.

By transforming its software development process from a corporate to a community-based project in the early 2000s, Mozilla radically redefined its rhetorical situation. It was no longer the product of a monolithic entity providing updates or innovations on an obscured schedule to a passive consumer base, but instead made the product *and* the process open and available—not just for consumption but for further development. However, this is not to suggest that Firefox's development is egalitarian, democratic, or even a meritocracy, even though the Mozilla website identifies itself as the last of these (Mozilla, n.d.a). Instead, the project has become a structured hierarchy of volunteers, from bug reporters to contributors to administrators; the administrative group determines which contributions at a given time, if any, will be added to the official code package for the program.

The role of hierarchical administration in regards to Mozilla's software programs has the potential to be extremely authoritative and, thus, restrictive of the otherwise "open" potential of community-based collaboration efforts. As Hamerly, Paquin, & Walton (1999) described the evaluation process:

> One of mozilla.org's most important roles is to draw lines as to what code is accepted and what is not. We must factor in a number of issues. First and foremost is merit. Is it good? [… Each of the projects comprising the Mozilla suite has] a designated 'owner.' That person knows the code best and is the arbiter of what should go in to that module and what shouldn't.

The administrator(s) of a particular program—for Firefox there are several—are able to influence the development of that program based on their anticipated vision for the program rather than on the merits of provided contributions. Even though the code may be weighed on whether it is "good," as noted above, that quality is defined by the project's "owner," a head administrator explicitly named to have control over the project. What does he or she value in regards to a particular line, or set of lines, of code? Who was involved in the process of naming an owner to the project, and what did the process resemble? Further, and perhaps of greatest interest to rhetoricians, what do these influences have on the types and means of action that are subsequently promoted through code and natural language discourse? While an entirely chaotic environment is not any more helpful to collaborative development than an overly authoritative one, it is important to recognize what sorts of constraints are put upon the development process by these decisions.

4.1.2.1: Software versioning systems and persuasion.

Code contributions to projects like Firefox or other open source programs, of varying scales and scopes, are commonly provided via software versioning (described briefly in the previous chapter), a system of different versions of a code artifact from one another in order to emphasize the specific line-by-line differences that exist in each iteration of the code files. Each developer maintains a different, but initially duplicate, copy of the program's "official" files and makes changes as he or she sees fit to various components of the program. Then, any changes which a developer might feel are worthy of inclusion in the official version of the program are returned to the project's management for review. If any of those changes are deemed acceptable, they are “committed” to the program, and the official version of the software code is updated to include those changes. Mozilla currently has almost one hundred and fifty developers listed publicly as members of its organization, participating in over three hundred distinct projects, through two major software versioning utilities.

Each of these utilities is fueled by a distinct general exigence, although the individuals contributing to either are spurred by their own sense of exigence and kairos. Mercurial, described first, serves as the primary tool with which the development of Firefox takes place: individuals engaging in the active, community-preferred (or at least leadership-preferred) work on the browser contribute their code to their companions through the Mercurial system. GitHub, described second, is a “social coding” website designed to make the versions of a program's code more accessible and collaborative. GitHub works as an archive of the contributions made through Mercurial, although there are numerous developers offering code changes through GitHub—effectively working outside of the “normal” workflow.

Mercurial is a “distributed” versioning system that enables developers with internet access to share their individual changes to a project's code by “pushing” those changes out to everyone else via a web-based repository (serving as a technological record-keeping intermediary) on the mozilla.org server as part of the organized Mozilla Developer Network (Mozilla, 2012a). Through Mercurial, which works very similarly to several other software versioning tools, each developer has his or her own *clone* of the project and makes changes to a personal *branch*—essentially a unique version of the code—and can request *pulls* to the central, official version of the project. A significant amount of discourse occurs between developers in this push-pull process, and some versioning tools refer to pull requests specifically as lively spaces for discussion and debate (GitHub, 2012); many developers are likely to suggest small (i.e. <20 lines of code) changes to specific files within the Mozilla project, and the administrators and various testers are tasked with determining the value and potential consequences of accepting each of those changes.

The counterpart to Mercurial is the repository website GitHub, based on the version control program git (Mozilla, 2012b). For its “Mozilla-Central” GitHub repository, which includes a significant majority of the code for the Firefox browser, Mozilla has twelve forks—potentially significant branches from the official repository “tree,” maintained and updated according to the individual schedules of the developers who initiated each fork. Several of those forks serve to draw in, hierarchically, changes made to other projects, through git or Mercurial, that effect the Mozilla-Central code and the programs it oversees in turn. GitHub serves as an accessible place for numerous *potential* developers who may not be (or who may not be interested in being) associated with the official Mozilla Developer Network. As one of GitHub's primary appeals, beyond its relative ease of use, is the number of users and repositories it hosts—at the time of this writing, over 2 million users were counted as collaborating on over 3.5 million projects (GitHub, 2012). It is thus safe to say that there is a strong possibility that Mozilla can draw the interest of random parties for occasional development through GitHub rather than through Mercurial, due to the former's relatively accessible web interface and visible archive of code contributions.

However, Mercurial and git are not the only tools used for Firefox development: Mozilla's Tinderbox browser-based test utility (http://tinderbox.mozilla.org) serves as an automated administrator for proposed code changes to and bug tracking for the browser's most up-to-date official code as maintained in the central Mercurial-based repository. Tinderbox's server regularly runs a set of tests for functionality as well as performance on the updated code to help determine whether or not it passes those tests (Mozilla, 2012c). In effect, Tinderbox acts as an influential administrator over proposed developments to Firefox (as well as to all the other Mozilla projects), an involved *co-rhetor* in the construction of the browser, as would-be contributors are constrained and overseen in their programming actions and activities by a “gateway” set of software programs. If a proposed code change does not comply with the goals of the Tinderbox tests, it is likely to be rejected by the human administration of the project, even if the goals of that code are valued and subsequently re-proposed, and potentially accepted, in another form. Code (in this case, Tinderbox) has a powerful rhetorical impact on Firefox's code (arising out of the project's development practices) through human developers modifying the latter, based on meaning (test results and their significances) provided by the former.

While GitHub similarly influences the practice of development through commit request capabilities (i.e. letting anyone propose code changes for review), Tinderbox is even more direct by having the versioning server *provide review* itself, implicitly passing judgment on the potential value of a code change. That is, the server becomes an active agent participating in the persuasive activities of software development related to the Firefox browser. The vast majority of the review "judgments" are reported mechanical failures, i.e. code unable to compile due to some error in one or more operations within the compilation process. What makes them noteworthy is that the output error messages provided as a result of Tinderbox's server testing is that they suggest to the human developer that his or her code —regardless of whether or not it works on his or her machine while testing that code individually—is faulty or otherwise unworthy of inclusion in the proposals for change to be discussed within and scrutinized by the more substantial Firefox development community. For those issues it deems critically in need of resolution, Tinderbox will display an animation of flames as if to suggest that the code (or its resulting compiled form) is "on fire" that must be put out before it hypothetically destroys the project. The Tinderbox system, in other words, serves as a kind of prototypical situated machine audience to whose specifications all Firefox code must successfully adhere. At the same time, Tinderbox is an active rhetor that promotes a specific form of action that it suggests should be undertaken by developers in order to improve upon the Firefox browser.

To an extent, a recognition of the Tinderbox server as the ultimate audience collapses much of the rhetorical potential of the general collaborative development on Firefox into a standardized, "sanitized" process in which the default product—the "vanilla" Firefox browser download emphasized on the program's website—is valued far beyond any of the customized, nuanced attempts to experiment with the software in which some developers or would-be developers might otherwise engage themselves. This is not to suggest that experimentation *does not* occur, but the atmosphere generated in part by the operation of the

Tinderbox server is one in which experimental play is not esteemed as highly as development towards universal productivity, to support the broadest of purposes, through the browser. Accordingly, the full range of rhetorical possibilities in regards to the code of the browser is not (yet) explored thoroughly at the level of the general Firefox development community.

4.1.3: Genres in code.

The development of OSS projects like Mozilla Firefox, like all the activities of collaborative and discursive communities, is social in nature: individual members contribute to a program's code base and deliberate, in and around that code, on how best to improve upon contributions from all manner of developer. However, the social quality of collaborative development is not limited to computational or efficiency-based optimization of code. Instead, the *activity* of participating in collaborative development serves to form and refine the nature of the development community itself. For groups that find themselves constantly and dynamically reforming through the waxing and waning enthusiasm of individual members of those groups, this social negotiation of community values and practices is key to ensuring that successful code and discursive practices alike remain fueled by contextual and kairotic factors.

Part of the negotiation of code-as-communicative-means involves a recognition of the variety of genres used by developers to persuade one another to act—in different ways and to different ends. Miller (1984) observed that genre serves as a means of rhetorical action that constitutes and reconstitutes a discourse community through its use. Among the implications

Miller provided for this social understanding of genre is the following: "[a] genre is a rhetorical means for mediating private intentions with social exigence; it motivates by connecting the private with the public, the singular with the recurrent" (1984, p. 163). In other words, genre provides a space for rhetors to understand the constraints and affordances available to them when interacting with particular audiences and identifying themselves as members of those audience communities. For communities making use of *genre systems*, what Bazerman (1994) defined as "interrelated genres that interact with each other in specific settings," (p. 97) we can see numerous purposes and persuasive efforts at work—sometimes harmoniously and sometimes in dissonant and competing ways.

For developers, the myriad genres of code, in-code comments, and meta-discursive commentary all function in ways that allow specific development communities, and individual members thereof, to establish their contemporary professional and community-based identities through generic performances. These performances simultaneously contribute to the confirmation of particular genres while also pushing back against them to reconstruct them according to changing social values and preferences. Code practices, such as those discussed in this chapter, are demonstrated not as unchanging mathematical constructions designed *solely* for computer technologies but instead as a fluid set of genres. Just like more discursive forms of communication, code genres are developed through the continued changing of logical and stylistic preferences that define "acceptable" compositions within specific communities. The processes that generate these compositions are significant components of these genres: it is not simply that multiple developers in an OSS community work together to create a program but that they communicate with one another *in specific ways* in order to create that program. The combination of in-line comments, email discussions, public reviews of code proposals, and the evolving structures and logic of code operations themselves. The community functions within this ecological system, and it thrives only when all members are able to access at least some of its components to contribute their efforts to the remainder of the community.

Murray, in his 2009 work on "non-discursive" rhetoric, asked a significant and relevant question for rhetoricians interested in computation, although he focused primarily on image as an alternative object of rhetorical study to writing and oration: how might rhetoricians understand the goals, appeals, and qualities of rhetoric when examining means of communication that are not primarily or explicitly discursive (that is, using natural, verbal human languages)? These concerns are key for any sort of rhetorical consideration of the variety of genres in which developers regularly work—especially those outside the bounds of what is conventionally recognized as discursive communication. For Murray (2009), the key was in understanding the generative qualities of a particular symbolic languages and systems; he pointed to the capabilities of images (and, implicitly, other meaning-making systems) as they allow rhetors to disregard explicit reasoning in favor of intuitive invention and play so as to present an argument to an audience (pp. 140-141). For code, the symbolic logic of computation itself can be acknowledged as a non-discursive system of symbolic meaning and action: as with image and other forms of communication, there is a barrier to "entry" in understanding expressions within that system. Further, persuasive activity is not just possible but *inherently present* in efforts to communicate through the system.

Inquiries such as these serve as points from which rhetoricians can begin to view more clearly the process of coding through a rhetorical lens, which in turn allows us to consider more fully and effectively both the act of coding and the range of expressions a particular act of coding enables and constrains. The means by which developers anticipate potential action, to be undertaken as part of the expression of a set of software algorithms, says a great deal about the expectations for action those developers incorporate into the very logic of their work. Similarly, the ways that users of that software have the potential to interact with it says just as much about how the values imparted from developer to user are recognized, accepted, or challenged as part of the software's use.

4.1.3.1: Demonstrations of innovation.

Firefox, as one of the major internet browser programs used around the world, has had numerous proposals to extend its capabilities beyond the simple rendering of HTML-related data. The majority of these are minor in nature, extending in specific and situational ways the functionalities of the browser, sometimes focused more heavily on code practices and style, while at other times the focus is on the expressive possibilities of the program's code. In either case, proposed contributions to the Firefox code base are often themselves further developed to determine just how beneficial, if at all, such a set of contributions might be to the larger community.

Among those features most often proposed and experimented with are tools *for the development of Firefox itself*, possibly since those tools are meant to be used by a relatively small population of users (the developers themselves). In such cases we can observe developers' persuasion upon other developers as the clearly identified audience. However, despite this focus the proposals possess a number of qualities which reflect practices related to large-scale development activities. For example, we can see in a recent adjustment to a development script the innovation upon an existing feature, as part of a tool called the Device Manager ADB, or Android Debug Bridge, designed for the Android mobile operating system. While the ADB is itself a relatively recent innovation (inspired by the rise of the mobile device OS), this adjustment to the code also introduced some new and potentially significant capabilities:

```
def pushDir(self, localDir, remoteDir):
    # adb "push" accepts a directory as an argument, but if the directory
    # contains symbolic links, the links are pushed, rather than the linked
    # files; we push file-by-file to get around this limitation
    try:
      for root, dirs, files in os.walk(localDir):
        relRoot = os.path.relpath(root, localDir)
        for file in files:
          localFile = os.path.join(root, file)
          remoteFile = remoteDir + "/"
          if (relRoot!="."):
            remoteFile = remoteFile + relRoot + "/"
          remoteFile = remoteFile + file
          self.pushFile(localFile, remoteFile)
        for dir in dirs:
          targetDir = remoteDir + "/"
          if (relRoot!="."):
            targetDir = targetDir + relRoot + "/"
          targetDir = targetDir + dir
          if (not self.dirExists(targetDir)):
            self.mkDir(targetDir)
      self.checkCmdAs(["shell", "chmod", "777", remoteDir])
      return True
    except:
      print "pushing " + localDir + " to " + remoteDir + " failed"
      return False
```
(Brown, 2012)

Originally, this code (written in the Python scripting language) enabled a developer to upload directories of files from the local hard drive to an external server. However, as noted by Brown in the comments at the beginning of the quoted excerpt, the code as originally written did not account for symbolic links within directories—links that pointed to other directories that may not be intended to get included in the set of data to be uploaded or that may not even exist (at all, or in an accessible location) on the remote server. Brown modified that code in order to work through each file and directory link individually, ensuring that all data being uploaded was an appropriate part of the call (i.e., removing irrelevant symbolic links from the operation). In addition, Brown included a line of code to modify each file's permissions as it is uploaded—a helpful task for developers but one that might compromise file security for non-developers, as he specifically set all uploaded directories (and their files) to be readable, writable, and executable by all users on a system.

Brown's innovation reflects broader practices in that he made use of multiple loops in order to ensure that the exact *sort* of outcome he anticipated was likely to occur successfully. Before Brown's proposed code change, there was no check to determine whether or not all files being uploaded were “valid” and not security risks. Brown's ordering of his operations persuades his colleagues to consider the implications of the ways that they attempt to construct software for their own, or others', benefit. It is not just that files are separated from directories but that each is checked against conditions ensuring its relevance to the attempted activity (the uploading of one or more directories and its files). Because they are not the same, Brown has distinguished them—but because they have similar qualities, Brown has repeated those tests to make sure that there is a lessened chance of an error occurring. In other words, Brown has positioned readability and clarity in intent over optimized computability so that the tool he worked on is not only improved by his contribution but that other developers can see how and why that contribution has a positive impact on the browser's development.

A second example provides insight into innovative developments based on the anticipated preferences of users when it comes to interacting with Firefox each time the program is opened. Originally, the browser was coded so that one of two events would occur when Firefox was started: if the user had saved a preferred home page, it would be loaded; otherwise, a default home page would be loaded instead (Walden et al., 2006a). However, this functionality was improved multiple times. One early improvement was for the function to recognize the possibility of a separate home page URI being set for each browser “tab” that a user might want to have open when the program starts (Walden et al., 2006b). A much more recent innovation made use of the browser's since-expanded capacity to provide a home screen in several lines added to the file, almost immediately above to the (unchanged) lines of code written four years earlier (Sharp et al., 2010). Excerpts from each of these innovations can be seen in comparison with the others in Table 4.1 below.

Table 4.1: Three iterations of Firefox startup home page code (2006, 2006, 2010)

```
var useCurrent = document.getElementById("useCurrent");
var chooseBookmark = document.getElementById("chooseBookmark");
var bookmarkName = document.getElementById("bookmarkName");
var otherURL = document.getElementById("otherURL");
[...]
if (bookmarkName.getAttribute("uri") == "(none)") {
  useCurrent.disabled = otherURL.disabled = true;
  bookmarkName.disabled = chooseBookmark.disabled = false;

  return "bookmark";
}

var homePage = document.getElementById("browser.startup.homepage");
if (homePage.value == homePage.defaultValue) {
  useCurrent.disabled = otherURL.disabled = true;
  bookmarkName.disabled = chooseBookmark.disabled = true;
  return "default";
}
else {
  var bookmarkTitle = null;

  if (homePage.value.indexOf("|") >= 0) {
// multiple tabs -- XXX dangerous "|" character!
// don't bother treating this as a bookmark, because the level of
// discernment needed to determine that these actually represent a
// folder is probably more trouble than it's worth
  } else {
#ifdef MOZ_PLACES
[...]
#else
[...]
if (bookmarkName.getAttribute("uri") == homePage.value)
  bookmarkTitle = bookmarkName.value;
#endif
  } (Walden et al., 2006a)
```

Table 4.1 Continued

```
var win;
if (document.documentElement.instantApply) {
  // If we're in instant-apply mode, use the most recent browser window
  var wm = Components.classes["@mozilla.org/appshell/window-mediator;1"]
                     .getService(Components.interfaces.nsIWindowMediator);
  win = wm.getMostRecentWindow("navigator:browser");
}
else
  win = window.opener;

if (win) {
  var homePage = document.getElementById("browser.startup.homepage");
  var browser = win.document.getElementById("content");

  var newVal = browser.browsers[0].currentURI.spec;
  if (browser.browsers.length > 1) {
    // XXX using dangerous "|" joiner!
    for (var i = 1; i < browser.browsers.length; i++)
      newVal += "|" + browser.browsers[i].currentURI.spec;
  }

  homePage.value = newVal;
} (Walden et al., 2006b)
```

```
syncFromHomePref: function ()
  {
    let homePref = document.getElementById("browser.startup.homepage");

    // If the pref is set to about:home, set the value to "" to show the
    // placeholder text (about:home title).
    if (homePref.value.toLowerCase() == "about:home")
      return "";

    // If the pref is actually "", show about:blank. The actual home page
    // loading code treats them the same, and we don't want the placeholder text
    // to be shown.
    if (homePref.value == "")
      return "about:blank";

    // Otherwise, show the actual pref value.
    return undefined;
  },

  syncToHomePref: function (value)
  {
    // If the value is "", use about:home.
    if (value == "")
      return "about:home";

    // Otherwise, use the actual textbox value.
    return undefined;
  }, (Sharp et al., 2010)
```

The basic functionality of this code is relatively stable across each text iteration, but the innovations incorporated by each set of developers offer together a valuable collection of persuasive arguments made *through* the code so that the functionality would not only be preserved but improved to coincide with other development efforts whose relation to this code is initially an unrealized potentiality.

The code draws originally on a simple but powerful ability—the saving of URI text strings as "bookmarks"—of which one could then serve as a startup home page (as a site likely to be visited frequently by the user). A separate URI could be used as a home page as well, or the default home page as a final choice (most notably http://www.google.com/firefox which allowed for a Mozilla-branded Google search). The first major update of the functionality involved expanding its use—fitting into a space commented on originally but not actually composed code-wise—to incorporate multiple tabs, separated in code by the `|` (pipe) character. This code makes use of a loop (when meeting the successful condition that `browsers.length > 1`) not unlike that demonstrated in the FizzBuzz examples in Chapter 2, iterating through each URI string in a user-provided list in order to load and display each appropriate website in its own tab. The second update builds upon *that* feature by prepending another set of functions to the relevant file, checking to determine whether the user-provided home page will be loaded or if the `about:home` screen will be loaded instead. `about:home` is part of a module that provides browser-specific abilities. For example, `about:plugins` provides information about the third-party plugins and extensions a user has installed in the browser. Similarly, `about:home` can either load a blank screen or the default home page, depending upon the user's preference—so even when no explicit *startup* page is defined, a user could still choose whether or not to load the default page.

Each of the innovations builds upon the previous iteration without a fundamental reworking or removal of that previous code. The development community is able to perceive the value of each added functionality, experimenting with the possibilities provided without necessarily demanding an overhaul of its purpose or of its code each time another contribution is proposed. Further, developments in one area of the project can often be easily included in the aims of another area, such as the browser-specific screen `about:home`. Even though the specific code added by each contributor differs somewhat in style, the generic purpose is nonetheless consistent: the community demonstrates its acceptance of participation by grafting new code onto existing code, experimenting with the boundaries of accepted code practices.

4.1.3.2: Normalizations of code practices.

Just as new features are constantly introduced by developers (new and veteran alike) and tested by the broader community, so too are the code texts comprising those features scrutinized and recomposed so that they adhere to the programming style prescribed by the community. Essentially, the generic conventions being pushed against as part of efforts to innovate are simultaneously locked in and cemented—even temporarily—by developers who feel comfortable with the forms of code-based communication they engage in with one another.

One such example describes the procedure by which a popup window is rendered and given focus. Lamouri et al. (2010) provided a Javascript module for the browser, serving as a means by which the program would show and focus on the popup window. Two years later, the *same feature* has been reworked so that the code's idiomatic style more fully reflects the procedural logic and rhetoric of Firefox development in a broader sense—see Table 4.2 below.

Table 4.2: Firefox popup removal code in Javascript, 2010 (upper) and 2012 (lower)

```
// If the user type something or blur the element, we want to remove the popup.
// We could check for clicks but a click is already removing the popup.
let eventHandler = function(e) {
  gFormSubmitObserver.panel.hidePopup();
};
element.addEventListener("input", eventHandler, false);
element.addEventListener("blur", eventHandler, false);

// One event to bring them all and in the darkness bind them all.
this.panel.addEventListener("popuphiding", function(aEvent) {
  aEvent.target.removeEventListener("popuphiding", arguments.callee, false);
  element.removeEventListener("input", eventHandler, false);
  element.removeEventListener("blur", eventHandler, false);
}, false);

this.panel.hidden = false;
this.panel.openPopup(element, "after_start", 0, 0);
```

(Lamouri et al., 2010)

```
// If the user interacts with the element and makes it valid or leaves it,
// we want to remove the popup.
// We could check for clicks but a click is already removing the popup.
function blurHandler() {
  gFormSubmitObserver.panel.hidePopup();
};
function inputHandler(e) {
  if (e.originalTarget.validity.valid) {
    gFormSubmitObserver.panel.hidePopup();
  } else {
    // If the element is now invalid for a new reason, we should update the
    // error message.
    if (gFormSubmitObserver.panel.firstChild.textContent !=
        e.originalTarget.validationMessage) {
      gFormSubmitObserver.panel.firstChild.textContent =
        e.originalTarget.validationMessage;
    }
  }
};
element.addEventListener("input", inputHandler, false);
element.addEventListener("blur", blurHandler, false);

// One event to bring them all and in the darkness bind them.
this.panel.addEventListener("popuphiding", function onPopupHiding(aEvent) {
  aEvent.target.removeEventListener("popuphiding", onPopupHiding, false);
  element.removeEventListener("input", inputHandler, false);
  element.removeEventListener("blur", blurHandler, false);
}, false);

this.panel.hidden = false;
```

(Akhgari et al., 2012)

The code is not *drastically* different between these versions, despite dozens of revisions and improvements to the code within the file in which these lines appear. However, the way each of the operations has been restructured is telling: the functionality added by Lamouri et al. (2010) was clearly valued by the community as a positive contribution to the program, but the *way* it was coded required normalization or standardization in order for it to be genuinely accepted. This is not to suggest that the code is now in any sort of "permanent" form—it may well continue to be revised for some time to come, especially as broader stylistic preferences in Javascript evolve.

So what exactly is happening in the code described in Table 4.2? The procedure is not extremely complicated: when someone closes a popup or completes a form in it, the popup will disappear from view (and/or provide an error message to the user if something unexpected occurs at any point during this process). What distinguishes the 2012 version of the code from the 2010 version is how individual events have been anticipated—most notably the shift from a general `eventHandler` object has been split into separate `blurHandler` and `inputHandler` objects, even while `addEventListener` and `removeEventListener` remain descriptive of general, catch-all functions. The `inputHandler` object is especially noteworthy in regards to what has been added, since it allows for developers and users to catch the reason(s) as to *why* a particular problem has arisen, most notably when a user is attempting to complete a form-based popup that closes unexpectedly. In addition, there is a validity check when `inputHandler` is called in order to determine whether or not its code *should* be executed, the condition beginning `if` `(e.originalTarget.validity.valid) {` (Akhgari et al., 2012). These possibilities are addressed obliquely in the initial set of comments in the 2012 version of the code:

```
// If the user interacts with the element and makes it valid or leaves it,
// we want to remove the popup.
// We could check for clicks but a click is already removing the popup.
```

(Akghari et al., 2012)

This extra-code explanation offers context as to reason for changes being made to the original version of the popup's closing functionality, but it also offers a judgment on that initial iteration: according to the original, the popup was removed when “the user type [sic] something or blur the element” (Lamouri et al., 2010). While the distinction appears minor, the user's “interaction with the element *and* mak[ing] it valid” (Akhgari et al., 2012, emphasis added) is significant—it signals to the developing community that an incomplete explanation of a particular task is potentially detrimental since it does not fully describe why relevant code attempts to achieve that task.

Interestingly, at least one extremely “non-industrial” practice has been left alone in the code. Specifically, the comment “One event to bring them all and in the darkness bind them,” present in both the 2010 and the 2012 version, refers to the ring of power in *The Lord of the Rings*, suggesting that the Firefox community maintains a still-thriving culture of geek humor. In addition, the comment also implies that a single, catch-all way to deal with events (as it precedes the general `addEventListener` function) may be preferable to more specific, customized methods—even though the 2012 code has been revised to incorporate specific `blurHandler`- and `inputHandler`-related operations. As a result, even though the "functional" code and informative commentary has been updated to reflect changes in community standards and preferences, the characterful commentary provided to describe *other* parts of code is left intact; it may indicate that "helpful" comments are a genre worth attending to, while incidental comments are not—a practice that in many ways seems to be the opposite of generic conventions for code, where the effectiveness of each line is significant. Ultimately, the community's efforts to norm its practices remain varied and inconsistent in execution.

Whether a developer interested in contributing code to the program, he or she must balance the desire to propose innovative code structures and expressive functionalities with a need to adhere to the socially acceptable stylistic and logical code practices that comprise Firefox development at that specific point in time. As participants in a genre system that makes use of oscillating social practices of innovation and normalization, the members of Firefox's development community are able to influence one another on how "best" to further the browser's capabilities. These persuasive efforts are not limited to larger-scale concerns, as many developers offer only minor changes to components of the browser's code. As a result, that code becomes a site ripe for examination of how specific rhetorical devices incorporated into code can influence development in particular ways.

4.1.4: Rhetorical devices in code.

Just as developers are compelled to demonstrate their larger-scale comprehension of how a program works (or should work) through the logical reasons underpinning their code decisions, so too are they expected to work effectively in a microcosmic fashion with the code they write. As demonstrated in Chapter 2 by the variety of means by which a developer could create a simple FizzBuzz program, the choices a developer makes at “smaller” scales—such as in individual functions or operations—rarely demonstrate objective superiority as much as they demonstrate the developer's approach to solving a particular problem and to articulating that solution through code. Further, it is rare that a developer attempts to explicitly persuade his or her audience through these code decisions. Instead, the potential intent behind his or her code contributions can and should be read as a set of rhetorical decisions as well as a set of computational decisions: it is not enough that code might *work* but that it is also understandable by, and persuasive to, an audience who needs to build upon that code to continue developing a program.

4.1.4.1: Anaphora, epistrophe, climax.

Those code functions and variables which provide critical operations (those valuable to a number of tasks) might be referred to, repeatedly, at crucial points in a larger body of code. To an extent, this sort of repeated call works as a kind of *anaphora*, the repetition of words or phrases at the beginning of statements, or as a kind of *epistrophe*, the repetition of final words or phrases for rhetorical effect. By repeating a particular idea in the same place, syntactically, a rhetor can draw attention not only to those repeated ideas, terms, or phrases, but also to the forms of argument that center on that repetition in order to induce audiences to act in various ways (often shifting subtly between each iteration of the repetition).

This notion of rhetorical repetition is closely related to the idea of *climax*, in which an argument's structure is based on significance—with more important ideas and points offered after less important ones. Climax is a key factor in computational procedure, because the order of operations within a given file or function impacts exactly how a set of data is calculated and used for particular purposes. While distinct functions and operations can be defined apart from one another, it is their combination in a certain order that facilitates the specific computational action(s) desired and anticipated by a developer. As a result, the persuasive activities engaged in by software programmers very often attend to the ways of reading code that suggest importance given to procedures based on their apparent anaphoric, epistrophic, climactic, or otherwise repetitive qualities.

This concern is especially important for a large-scale OSS program like Firefox, since it involves work on dozens of interlinked modules each of whose code needs to set itself apart from the others. However, each of these same modules must also maintain a stylistic form close enough to the others to make it capable of being modified and improved upon by an interested party who might have worked on some other component of Firefox code. While the code can never reach a scalable "fractal" state (wherein the same general structures are repeated at different scales of code), there are nonetheless observable efforts by developers to persuade one another to implement and maintain particular forms of procedural repetition as part of an effort to suggest the use of some coding paradigms over others.

One such example of repetition within the code for a single module is the early work composed a decade ago for a spam filter in the Mozilla suite's email and news program (which has since become the program Mozilla Thunderbird). Dan Mosedale and Peter Vande Beken contributed the initial code to the project, making use of a function (among others) called `processNext()` to iterate through the list of messages to be read and dealt with by a user. Technically, there were multiple `processNext()` functions, each working similarly but with a different focus: one to move between folders, one two move between messages within a folder, one to mark spam messages, and one to mark spam folders. Each function in which some variation of `processNext()` resided called `processNext()` at the end of its own operation in order to compute whether or not `processNext()` would need to be run again. The shortest of these variations is the following:

```
function markFolder(aSpam)
{
  function processNext()
  {
    if (messages.hasMoreElements()) {
    // Pffft, jumping through hoops here.
      var message = messages.getNext().QueryInterface(nsIMsgDBHdr);
      mark(message, aSpam, processNext);
    }
    else {
      gJunkmailComponent.mBatchUpdate = false;
    }
  }
```

```
    getJunkmailComponent();
    var folder = GetFirstSelectedMsgFolder();
    var messages = folder.getMessages(msgWindow);
    gJunkmailComponent.mBatchUpdate = true;
    processNext();
  }
```
(Mosedale & Vande Beken, 2002)

The function in which a given folder is determined to be spam calls `processNext()` multiple times when it is run. First, it is defined as a sub-function of the `markFolder()` function. Here it is not *run* but its operational structure is established so that it can be run when called elsewhere in the code. Second, it is referred to as part of the `mark()` function inside `processNext()` itself—meaning that it provides the data from its own execution to another function that can make use of it. Second, and more importantly, it appears as the last line of the `markFolder()` function—the same place it appears at the end of the other functions in which some variation of `processNext()` is defined. In essence, every time `markFolder()` is called, `processNext()` will run as the final and climactic function to ensure that it can potentially be run again as needed; its name even suggests this sort of forward progression that builds up as it proceeds. Since every component function of the spam filter relies on a version of `processNext()`, its role as perhaps the *most significant* part of the filter code is made clear through its repeated calls as well as in its position as the final function executed within each part of that code.

While code does not necessarily *demand* an epistrophic or climactic approach to its computational operations, it is nonetheless true that the procedural structure of its algorithms generally relies upon an accumulation of logical complexity and activity from each line to the next. In other words, an early computation influences subsequent computations. As a result, "late-stage" operations are often the most complex or significant sets of computation as they have the potential to work with the results of earlier operations. However, much of this structure arguably is from a desire by developers to communicate an intended functionality to themselves or collaborators; this approach is what Knuth (1992) has called "literate programming," a means by which programmers clearly articulate what their code does through the code itself. Literate programming stands in stark contrast to most programming whose clarity is defined by extra-code documentation like comments, specifications, or discussion in other forums. While most code's readability is influenced most explicitly by the names chosen for specific functions, variables, and objects, there is an implicit argument made by a developer for a particular logical structure as presented to readers through the code—such as the variations in "FizzBuzz" discussed in Chapter 2. This implicit argument is evaluated as much on its ability to be understood by developers as its ability to compute successfully.

A relevant and interesting quality of object-oriented languages like C++, Java, Ruby, and others is that the *definition* of particular functions and objects can occur outside of linear procedure—declaring what a block of code *does* is separate from calling that code (having it actually compute as part of the executing program). Note, for example, that the `processNext()` function included within the function `function markFolder(aSpam)` above, while called as the final component of the code block, is the first piece of code *defined* within it—a crucial bit of information that clarifies for a reader just what this specific version of `processNext()` will do in this context (as distinct from `processNext()` for the other mail-related activities that might take place when the program is used.

The structure of computational procedure—where early operations build upon one another to deliver potentially complex subsequent operations—suggests an implicit recognition of significant *ordering* for rhetorical purposes, where a developer indicates an important set of concepts to his or her audience at particular steps in an algorithm. This suggestion may be for creative or professional purposes, stressing a paradigm of practice whose impact extends beyond the specific instance under scrutiny at any given moment. However, such practices are not limited to the placement of significant operations and procedures (in climactic, anaphoric, or epistrophic senses) but also the idea of repetition itself: when should a given function be called? When should a set of operations be written multiple times for similar or distinct purposes? These considerations are addressed below.

4.1.4.2: Exergasia.

Just as repeated, or closely similar, arguments can provide both rhetor and audience with an understanding of the range of possibilities available to either through those arguments, so too can repetition serve to cement the suggested necessity for a particular rhetorical approach. Repetition in code can offer developers with an understanding of preferred stylistic practices by associating together multiple separate functions or sets of operations. Rhetorically, this can be considered a type of *exergasia*, the repetition of a significant idea across multiple forms of expression. For software programs that consist of hundreds of thousands of lines of code, exergasia is a helpful device that works to instruct developers on how other developers have determined code structures should work. Simply put, it provides a procedurally rhetorical engagement, to use Bogost's (2007) term, with the code's anticipated functionality—both as a component of the larger program *and* as an entity that persuades audiences to act in relation to its facilitated activities.

For a massively collaborative OSS program like Firefox, exergasia is a powerful tool by which developers signal to one another how particular procedures should be constructed and executed. By making use of explicit repetition in order to accomplish multiple related tasks, a developer can suggest that the operations and syntactical structures of the repeated code are valuable by way of both the frequency of such structures' repetition and the relations perceived to exist between each iteration of the code. Similarly, the *method* of repetition is critical to an understanding of code as rhetorically powerful communication. Given the ability for code to loop iteratively through a set of operations, it is possible for repetition to occur *conceptually* but not *explicitly* in the statements that comprise the looping code. Looping is often viewed as an elegant way to describe iterating code, so when it is *not* used, we are presented with an opportunity to examine why more “conventional” repetition is implemented as well as what its implementation can suggest to us about the program in a more general sense.

Object-oriented programming languages have the potential to make especially interesting use of exergasia through the instantiation of unique objects whose code is, in effect, a repeated set of variable definitions and functions across all members of a specific class. That is, an object is a bundle of data with properties that are shared across multiple objects but whose *use* of those properties is distinct from other objects' use thereof. This concept has its root in Plato's theory of forms; the ideal version of a thing is reflected, somewhat poorly, in real interpretations of that ideal. “Human” might be considered an object class, and every human is thus an instantiated object of that class: we each (generally speaking) have two eyes, a nose, two arms, and so on, just as we are covered in skin and possess the same assortment of skeletal bones and internal organs. However, no two humans (other than twins) could be said to be *identical* despite these shared features. As a result, we can discuss activities and procedures in which humans can engage, thanks to the set of capacities that extends beyond the scope of a single individual, but the specific activities that are accomplished are likely to differ between individuals' distinct executions thereof.

The idea of a human as a helpful example “object” for the purposes of object-oriented programming is so pervasive that it stands in as a demonstration in Mozilla's developer documentation for contributing to the Firefox project (and other relevant projects). Specifically, documentation authors Trasviña et al. (2012) describe the ways in which object-oriented languages like Javascript enable practices of iteration and repetition alike through the *potential* constructed through object classes, using “person” as the most accessible example by which to model object-oriented programming:

```
function Person(gender) {
  this.gender = gender;
}
Person.prototype.sayGender = function()
```

```
{
  alert(this.gender);
};
var person1 = new Person('Male');
var genderTeller = person1.sayGender;
person1.sayGender(); // alerts 'Male'
genderTeller(); // alerts undefined
alert(genderTeller === person1.sayGender); // alerts true
alert(genderTeller === Person.prototype.sayGender); // alerts true
```

(Trasviña et al., 2012)

In this example, a formal class has been established (`Person`) and a specific person has been defined based upon the properties of that class (`person1`). Any adjustments made to the larger class will affect the `person1` object, and any number of objects can be instantiated to serve as distinct entities that are nonetheless “repetitive” in terms of the code functions they share with `person1`. In this sense, the relevant code is simultaneously repeated across all instances of the class *and* expressed in potentially unique ways, as each iteration of the class differs from most, if not all, of the others. The argument provided through the use of exergasic code through object-oriented structures can implicitly and explicitly call attention to the modular and repeatable nature of the functions and operations called to help achieve certain goals.

It is no coincidence that “person” serves as the go-to example for such a discussion. By equating “personness” with “objectness” to explain how object-oriented code works, Trasviña et al. (2012) suggest that the construction of code objects is as full of symbolic meaning as any other means by which we might consider and constitute the human form. To an extent, this example provides its own exergasic demonstration of the variety of unique qualities that might be possessed by any individual person—qualities that only emerge through the activity of computational action. To state it bluntly, we are what we do, and we are constrained in what we do by what we *can* (and cannot) do, sets of affordances defined individually and socially, not unlike the sorts of repetitive rules outlined in the `Person` class example.

For an example inside Firefox's production code base, Chevalier (1998) provided an early series of code blocks to compute the size parameters for the minimum and maximum width and height properties of various components of the Firefox browser. Javascript, like many code languages, has a pair of properties called `width` and `height` that can belong to certain types of objects and which store the rendered width and height values (in pixels) of entities drawn on a computer screen. However, this only provides one value for each dimension—for those components of a program that calculate their own size as derivatives of other objects' `width` and `height` were implemented, customized functions are necessary to make and store those calculations, and Chevalier composed four such relevant function blocks. Because these blocks worked *almost* identically, Chevalier provided a uniform structure across all four of the methods needed to compute their values (i.e. minimum width, minimum height, maximum width, and maximum height). One of the code blocks appears in his initial text as follows:

```
nsStyleUnit minWidthUnit = mStylePosition->mMinWidth.GetUnit();
if (eStyleUnit_Inherit == minWidthUnit) {
  mComputedMinWidth = aContainingBlockRS->mComputedMinWidth;
else {
  ComputeHorizontalValue(aContainingBlockWidth, minWidthUnit,
                     mStylePosition->mMinWidth, mComputedMinWidth);
```

(Chevalier, 1998)

This block is comprised of two parts. The first is a variable declaration that sets the minimum width unit size to be used for comparative purposes. The second is a conditional statement that determines what the minimum width of a browser component should be, based upon the size restrictions imposed upon it by any "containing block," or parent-level component in which the currently-computing component resides. The associated size blocks differ only in the specific variables being called by the relevant code—for example, height-related blocks look for "vertical" rather than "horizontal" values, and maximum-related blocks look for "max" rather than "min" values.

Because the four blocks are so close in resemblance as well as in proximity to one another, there is a relatively clear suggestion that they have related functional purpose. Given the size of Firefox's development community, this suggestion is powerful: it implies a particular stylistic scheme (namely, that related code should look and function alike) *and* that this is the scheme preferred by the community at large, the normalized and industry-aligned practice. As demonstrated earlier, such preferences are constantly changing—hence the `width` and `height` properties being shifted from custom variables to object-oriented properties. Nonetheless, the kairotic appeal of newly-introduced code can energize its use for some time before critical scrutiny is applied to the specific means by which that code is constructed.

4.2: Conclusions

Just as rhetorical action is ever present in discussions about any human activity, so too is it demonstrated *through* the activity of code production. The social practices that developers engage in as part of their efforts to persuade one another to code in certain ways serve to influence them—as communities and as individuals—to participate in particular *types* of development, for particular ends. Because code, like other forms of language, serves to describe *more* than what it literally states, the variety of rhetorical strategies and devices available to developers in code is relatively astounding. This is significant in that developers can, and do, persuade one another in implicit and explicit ways to accept the practices they suggest through the texts they produce.

Mozilla Firefox provides a locus for an in-depth analysis of persuasive efforts in code across a massive community of thousands over fourteen years of collaborative development. While much of the specific code produced reflects industrial as well as individual stylistic preferences and trends for invention, it nonetheless demonstrates a varied range of innovative attempts by numerous programmers to engage their colleagues (as audiences) in creative and highly suasive ways. These attempts can be recognized at relatively large scales, such as in how innovative experimentation and normalizing revision occurs over time. However, they also occur at small scales, such as when particular logical operations make use of climactic or epistrophic structures in order to lead a developer audience to compose code in alignment with those structural paradigms. As a result, we are able to witness a dynamic, continually-changing ecology of development practice and persuasion whose components also shift and develop over time.

In the following chapter, I will turn from exploring specifics in code to considering how rhetoricians might make effective use of assessing rhetorical appeals made in and through computation and code. The ability to assess persuasive efforts in these modes of communication can help rhetoricians interested in code in establishing a more solid relationship between rhetorical code studies and traditional research in rhetoric. Finally, I will offer suggestions on how scholars of rhetoric and code alike can build upon the work laid out here for the further development of rhetorical code studies beyond the scope of possibility discussed so far within this text.

CHAPTER 5: CONCLUSIONS

Upon undertaking this project, I hoped to take advantage of the current opportunity for a critical examination of the relationship between the rhetorical possibilities of algorithmic computation and the computational qualities of rhetoric. Scholars interested in the means by which digital technologies enable and constrain particular ranges of action could similarly offer novel insights into our understanding of rhetoric by considering how the relationship between rhetoric and computational logic can offer us insight into the workings of both. It is not merely enough to identify rhetoric "as" computational or computation "as" a form of rhetorical communication; the recognition of the one as part of the other is meant to serve as an introductory foray into experimentation with expressions of the potential action(s) that can be undertaken as a result of this knowledge. By teasing out some of these possibilities through identifying the cultural influences on and implications of algorithmic computation—in an abstract sense, in specific cases, and in the writing classroom—I hope to provide a starting point from which rhetoricians can begin incorporating computational technologies more fully and naturally into the body of objects of serious rhetorical study and composition.

In this chapter, I articulate the broad strokes of the argument made thus far in the establishment and defense of a rhetorical code studies. In addition, I discuss the future of the field, identifying some existing means of assessment related to the critical and rhetorical inquiry of software code. Finally, I discuss how those means may be developed more fully so that they may support a broader and deeper expansion of rhetorical code studies. The field I have proposed is not meant to be read as a universal solution to the questions of digital rhetoric or the study of software, but instead as a space in which to engage issues emerging from convergent inquiries undertaken by scholars in these disciplines.

5.1: Rhetorical Code Studies So Far

As discussed in my first chapter, as a field, rhetorical code studies can best be defined as the convergent space shared by the disciplines of rhetoric, software studies, and critical code studies. While there has been significant and influential scholarship in each of these areas that conceptually overlaps with the scholarship in each of the others, to date, little has occurred in terms of explicit cross-disciplinary acknowledgement or engagement between these disciplines. This lack of boundary crossing is particularly notable in regards to rhetoric, as a number of critics in recent years have expanded our understanding of “digital rhetoric” as an area worthy of serious study. However, most rhetoricians interested in digital media have focused on the end-user interfaces (i.e., software programs) most commonly used for purposes of invention and communication rather than on the software code languages and “hidden” interfaces that facilitate subsequent end-user actions. A turn to *code* enables rhetoricians and critics of software to explore the possibilities of meaning-making among software developers as well as the meaningful interactions they facilitate for broader sets of users.

At the center of a rhetorical code studies is the algorithm, and specifically the algorithm as a way to understand the creative processes we engage in regularly as part of our humanistic activity. While algorithms are conventionally thought of in terms of engineering, mathematics, and computer science, algorithmic procedure has its roots in the day-to-day activities humans have engaged in for millennia. Building upon this history of algorithmic procedure as a description of fundamentally creative processes, I explored in Chapter 2 the relationship between algorithms and *enthymeme*, the central mechanism with which rhetorical arguments are delivered to audiences. Specifically, an enthymeme functions algorithmically in that it implicitly demands some computation on the part of an audience—the completion of an incomplete syllogism. This demand engages that audience in the rhetorical act, but only so long as the audience recognizes and performs the computation of the rhetorical algorithm offered to it.

Conversely, algorithmic procedure as present in (and communicated through) *software code* makes use of enthymematic reasoning to anticipate how that code will execute as part of a user's activities. In other words, developers provide implicit arguments, using what Lanham (2003) referred to as *tacit persuasion patterns*, to persuade other developers to engage in specific styles of development, writing code that functions in particular ways. The logical structures of code enable multiple means of responding to a given exigence—so what becomes important is how developer audiences interpret and complete the code-based enthymemes provided by their colleagues. This importance can be viewed across multiple scales of code development, from individual function logics to larger concerns of data iteration (as demonstrated in that chapter through the specific examples of the FizzBuzz test, the quine, and the HashMap concordance).

Conventional forms of discourse play a significant role in code-related rhetoric, as software developers converse with one another about both their preferred means of accomplishing specific tasks in code and their arguments for why other developers should follow similar approaches to coding. This sort of discourse can be most easily observed in the discourse of large open source software communities, in which hundreds or thousands of developers engage in collaborative software program development over extended periods of time. These developers often have varied levels of expertise and familiarity with the relevant programs and languages used to make those programs, so the conversations that take place within a given community provide helpful insight as to how particular developers attempt to persuade their fellow contributors. In Chapter 3, I examined the discourse of the development community for the Mozilla Firefox web browser. An extremely massive and popular open source software program, Firefox has been collaboratively developed for fourteen years by thousands of professional and amateur programmers. As a result, the range of conversational topics, and the range of rhetorical appeals used in relation to those topics, is rather broad—even when considering the relative scope of discussion “narrow” (i.e., focused on the development of a single program). Perhaps unsurprisingly for rhetoricians, the Firefox developers engaged in practices making use of appeals to ethos and pathos as much as, if not more than, logos, suggesting that decisions about development practices are not focused so much on computational efficiency or optimization as on personal preferences and stylistics as well as group dynamics.

In Chapter 4, these considerations were extended into the code texts and development practices themselves. I examined several particular types of composition process and rhetorical strategy present in Firefox's code at various points in its history (as well as in its current form). Each of these examples showed a fundamentally rhetorical approach to writing in and through code, in regards to composing both for a developer (colleague) audience and *with* colleague collaborators on a shared set of texts whose constraints influence the work—rhetorically meaningful code construction—undertaken by involved members of the community. For Firefox, as with most collaborative development projects, this involves *additive* practices of code composition, reflecting the back-and-forth of conversational discourse (wherein one speaker responds to, but does not eradicate, the statements of others). This fundamentally rhetorical quality of discursive communication as demonstrated *in code* is extremely significant, as it allows scholars to observe how software code languages facilitate persuasive activity between human beings and not simply mechanical processes for, or in, computer technologies. I pointed to examples of rhetorically powerful arrangement, such as anaphora, climax, and exergasia, as recognizable persuasive strategies that imply specific ways of solving relevant problems and manipulating data as the optimal means of achieving those goals. While it would be inaccurate to claim that most (if any) of the Firefox developers were consciously attempting to persuade with these strategies, their *use* nonetheless has implicit persuasive effects on the developer audiences who engage those texts and practices.

Ultimately, this examination of code and code-related discourse as rhetorical and significant forms of meaning-making serves to demonstrate the importance of rhetorical code studies for twenty-first century studies of rhetoric and digital media. In particular, the potential for code to facilitate and constrain ranges of action reflects the dynamics of rhetorical invention and delivery—albeit in a set of forms that have been, to date, unfortunately under-examined in relation to the significant impact digital technologies (and thus developers' decisions) have on our day-to-day activities. Such an examination would benefit rhetoricians, software critics, and code critics alike: just as we can understand more clearly the cultural influences on, and consequences of, software practices, so too can we explore more fully how we attempt to communicate meaningfully with one another in and through those practices. In short, we have an opportunity to approach investigating the ranges and types of actions we attempt to induce in various audiences (of developers as well as of users) for particular purposes.

5.2: Assessing Computational Action

If the goal of rhetoric is to facilitate action, and if this action is made possible by the inherently procedural nature of rhetoric, then computation—which similarly operates through procedural expression—is capable of similarly facilitating some form of action of value and interest to rhetors *for rhetorical ends*. This is not a logical given (since computation may not *always* be used for such purposes), but it is possible to recognize that, and how, computation is action-oriented toward many of the same contingent and situated ends as rhetoric.

One might argue, to a relatively accurate degree, that computation and rhetoric differ in that the former, unlike the latter, cannot engage in any sort of explicitly discursive give-and-take with an audience, and neither can the logic of a computational statement be debated by a machine (it instead will either be accepted as valid or refused as invalid). However, the structure and intended effects of both a computational procedure and a rhetorical procedure are often closely aligned if not parallel in nature. In essence, this is because computation does not occur without context: there is a reason for the expression of a given procedure, and that reason is often to accomplish some *meaningful* outcome for various explicit and implicit purposes. As demonstrated in previous chapters, many professional software developers recognize that their work has these meaningful qualities, but they rarely engage in substantive discussion thereof due, in part, to a lack of engagement with a humanistic (and specifically rhetorical) vocabulary that would help clarify how those procedural development practices function in these ways. Scholars interested in the rhetoric of code can help bridge this gap between critical analysis and pragmatic practice, but it requires an ability not only to translate rhetorical principles to professional and public audiences but to help those audiences *assess* the possibilities of rhetoric communicated through code.

Bogost (2007) addressed such a contextual concern as part of an examination of the potential ways to assess procedural rhetoric, especially in relation to video games. For Bogost, assessment was crucial because it "always requires an appeal to an existing domain.

An assessment equates one form of symbolic action with another form of symbolic action through some mediating measurement" (2007, p. 322). In other words, the meaning of a particular set of behavior is given a second set of meaning(s). Bogost specifically described the assessment of game play as "a form of procedural symbolic action […] compared with desirable behavior within an institution, via material measurements like written texts or job performance" (2007, p. 323). For computational action, rhetorical assessment provides a means of outlining the suasive influence of particular algorithmic procedures on human behavior as well as on the construction of machinic behavior (as a result of influenced human activity). For rhetorical action, computational assessment offers a perspective focused on the structure(s) of anticipated activity to be executed through and as a result of a given attempt at meaningful suasion. Bogost (2007) suggested that "procedural rhetorics can […] challenge the situations that contain them, exposing the logic of their operations and opening the possibility for new configurations" (p. 326). In other words, examining and assessing algorithmic procedures can not only shine light on how they work, or toward what ends they function, but also how other suasive procedures might be constructed for other purposes and audiences.

But by what metrics can the aforementioned types of assessment be evaluated? It is admittedly easier for rhetoricians to consider the ways in which computational procedures—especially as code texts—might be read as meaningful communication; Burkean dramatism, as described by Burke (1962), even offers one specific means of approaching computation with an algorithmically-oriented school of criticism. Reading code as rhetorical text (while using an interpretive lens such as Burke's pentad) offers new possibilities of understanding symbolic action communicated through forms that have yet to be explored, mostly due to the long-standing definition of code as machine-focused and non-meaningful instructions. Admittedly, Burke argued that meaning was limited to human communication (with nonhuman activity instead reflecting non-symbolic “motion” after Hobbes; see Burke, 1962, pp. 135-137), but scholars in more recent decades have addressed the possibilities of symbolic action with more nuanced consideration. The pentadic ratio described by Burke even functions as a kind of algorithmic procedure not unlike the classical enthymeme, with a Burkean critic reaching an interpretive conclusion based upon the pairing of certain dramatistic elements related to a given rhetorical act or event.

Another useful framework for rhetorically computational assessment is Shipka's (2005) task-based model for evaluating multimodal composition, since it emphasizes the processes and procedures of rhetorical communication through multiple means, as well as modes, of meaning-making. When using Shipka's structure, “questions associated with materiality and the delivery, reception, and circulation of texts, objects, and events are no longer viewed as separate from or incidental to the means and methods of production, but as an integral parts of invention and production processes” (2005, p. 301). Shipka's framework can apply to code and the technologies that facilitate it as easily as to any other form of composition; within such a structure, the goal is not to achieve one unified end but to allow students to discover their audience and purpose for a given task as well as the optimal means of achieving that purpose. As Shipka (2005) observed, making use of any and all "*operations*, *processes*, or *methodologies* [… and] the *resources*, *materials* and *technologies* that will be (or could be)" [sic] used as part of a given suasive activity is integral to the skillful use of rhetorical principles for persuading an audience to engage in some form of action. Accordingly, a rhetor's awareness of, and ability to reflect upon, his or her employment thereof is a significant component of any effort to assess the quality and success of using computation successfully for rhetorical purposes.

Focusing on assessment via the *tasks* involved in an act of rhetorical composition is also effective since a task-based model calls attention not only to the procedural nature of the invention process but, more importantly, to the explicit evaluation of the *actions* a rhetor means to facilitate through his or her suasion. That is, while any assessment of rhetorical composition is going to include an examination of how and why a rhetor attempts to communicate with a given audience, a task-based assessment emphasizes the rhetor's awareness of the subsequent action(s) that he or she attempts to achieve or otherwise influence through the suasive act.

Such a model is aligned with the goals of activity theory and its computational structure, as well. In activity theory (AT), a set of subjects and their object emerge through the course of undertaking and accomplishing (or, at least, attempting to accomplish) a particular activity. AT has primarily been applied to human-computer interaction, but its fundamental principles could just as easily be applied to the rhetorical practices of software code development as well as of its use. Christiansen (1996) has argued that

> *activity* [is] the term for the process through which a person creates meaning in her practice, a process we can neither see or fully recall but a process that is ongoing as a part of the participation in a community of practice. Activity is a process that we can approach by unfolding the task as stated within the community of practice and the objectified motive of the activity[.] (p. 177)

While the focus in Christiansen's assessment is clearly on *process*, it is a process meant to accomplish a given set of *tasks*, each of which in turn has a rhetorical goal and a computationally-informed structure that has led to the expression of the overall activity (i.e., persuasion). AT offers a framework in which to explore the rhetorical ecologies and genres of particular development practices, helping scholars and developers alike to understand how all components of an activity contribute to its undertaking and achievement.

In addition, emphasizing tasks as a fundamental component of rhetorical invention allows for the assessment of the procedural logic that supports a particular attempt at meaning-making. How might an audience respond to, or build upon, a given argument? What sorts of affordances or constraints has the rhetor incorporated into his or her communication so as to influence any potential responses? In answering these questions, assessors need to understand the algorithmic operations used by the rhetor to maneuver from inventive potential to realized result. Without such an evaluation—if the "computation" of the rhetorical situation is ignored—then we lose the opportunity to explore the possibilities that alterations in the rhetorical algorithm's expression might have produced: how different variables (e.g. audience, purpose, modes of communication, tone of message, etc.) would influence the outcome, how certain conditions (e.g. how skeptical an audience initially is of the rhetor's position) might modify the restrictions upon specific appeals being made, and so on. A rhetorical code studies demands not just looking at computer code as text but also evaluating how rhetorical communication of any type employs computational procedure in order to succeed.

5.3: A Future for Rhetorical Code Studies

This project has attended to the need for critics of rhetoric, software, and code to engage and understand the rhetorical qualities of, and means of persuasion for, meaningful communication through software code languages. Beyond arguing for code as a form of rhetorical activity through extensive discussion of general strategies and specific examples in the case of the Mozilla Firefox browser, I have offered in this chapter several suggestions as to how interested critics might move forward with the assessment of code through various relevant theoretical frameworks. While these suggestions help us take a step forward towards a fuller rhetorical code studies, they do not exhaust the possibilities for scholarly inquiry into the development practices and processes that serve as the foundation for digitally mediated action across populations.

In conjunction with the methods of assessment described above, we can ask significant questions about the intended, and perceived, efforts at suasion described in and through code. For example, how might we begin applying rhetorical principles to existing software programs and practices so that we might better understand the complex computational-rhetorical processes in which we regularly engage? Such an undertaking involves not simply identifying particular operations, methods, or function structures as possessing specific rhetorical qualities but also recognizing the interplay between lines of code, social interactions between developers (if there are multiple developers involved), bureaucratic procedures influencing particular developers' contributions to a project, and the software facilitating development activities—to say nothing of the situations surrounding and informing the *use* of that program once it is released to public audiences.

For scholars whose research interests focus on the various ways in which discourse, interaction, culture, and digital technologies all overlap and influence each other, rhetorical code studies can provide a foundation upon which to build a more critically- *and* technically-oriented approach to the study of these convergent forces. While it may not be necessary for all scholars of rhetoric or software to consider the role of the other field as part of their work, rhetorical code studies offers a means by which both parties can extend their investigations in dimensions that would otherwise remain black-boxed, unexplored, and otherwise unacknowledged—in other words, a continuation of scholarly traditions that have left us unprepared and unable to address the significant impact of code on the digital programs and systems we use, for rhetorical purposes, every day.

REFERENCE LIST

Abdullah, R., et al. (2009). The challenges of open source software development with collaborative environment. *2009 International Conference on Computer Technology and Development*. Vol. 2. Kotakinabalu, Malaysia. pp. 251-255. Retrieved from IEEE Xplore.

Akhgari, E., et al. (2012). Bug 829870 – Only draw in the title bar of private windows which are browser windows on Mac; r=gavin. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/88ce997385044499f1781dcf7b80a1a969e282d8/browser/base/content/browser.js

al-Khwarizmi, Abu Abdallah Mohammed ben Musa. (Trans.) (1831). *The algebra of Mohammed ben Musa*. (Trans. F. Rosen). London: J.L. Cox. Retrieved from http://books.google.com/books?id=3bNDAAAAIAAJ

Apache Software Foundation. (2012). How the ASF works. *The Apache Software Foundation*. Retrieved from https://www.apache.org/foundation/how-it-works.html

Aristotle. (trans. 2007). *On rhetoric: A theory of civic discourse*. (G.A. Kennedy, Trans.). Oxford: Oxford UP.

Arola, K. (2010). The design of Web 2.0: The rise of the template, the fall of design. *Computers and Composition, 27*(1), 4-14.

Ballentine, B. (2009). In defense of obfuscation: Questioning open source and a new perspective on teaching digital literacy in the writing classroom. In S. Westbrook (Ed.), *Composition & copyright: Perspectives on teaching, text-making, and fair use* (pp. 68-89). Albany: SUNY Press.

Barker, S. (2000). The end of argument: Knowledge and the internet. *Philosophy and rhetoric, 33*(2), 154-181.

Bazerman, C. (1994). Systems of genres and the enactment of social intentions. In A. Freadman & P. Medway (Eds.), *Genre and the new rhetoric* (pp. 79-101). London: Taylor & Francis.

Bénabou, M. (2007). Rule and constraint. In W. Motte (Ed.), *Oulipo: A primer of potential literature* (pp. 40-47). Champaign, IL: Dalkey Archive Press.

Benson, T.W. (1996). Rhetoric, civility, and community: Political debate on computer bulletin boards. *Communication Quarterly, 44*(3), 359-378.

Berge, C. (2007). For a potential analysis of combinatory literature. In W. Motte (Ed.), *Oulipo: A primer of potential literature* (pp. 115-125). Champaign, IL: Dalkey Archive Press.

Berlinski, D. (2000). *The advent of the algorithm: The 300-year journey from an idea to the computer*. San Diego: Harcourt.

Berry, D. (2011). Iteracy: Reading, writing and running code. *Stunlaw: A critical review of politics, arts, and technology*. Retrieved from http://stunlaw.blogspot.com/2011/09/iteracy-reading-writing-and-running.html

Bitzer, L.F. (1959). Aristotle's enthymeme revisited. *Quarterly Journal of Speech, 45*(4), 399-408.

Bitzer, L.F. (1968). The rhetorical situation. *Philosophy and Rhetoric, 1*(1), 1-14.

Black, P.E. (2007). Algorithm. In P.E. Black (Ed.), *Dictionary of algorithms and data structures*. Retrieved from http://xlinux.nist.gov/dads//HTML/algorithm.html

Blythe, S. (2001). Designing online courses: User-centered practices. *Computers and Composition, 18*(4), 329-346.

Bogost, I. (2007). *Persuasive games: The expressive power of videogames*. Cambridge, MA: MIT Press.

Bogost, I. (2008). Platform studies. *SoftWhere 2008*. Retrieved from http://emerge.softwarestudies.com/files/05_Ian_Bogost.mov

Bogost, I. & Montfort, N. (2009). Platform studies: Frequently questioned answers. *Proceedings of the digital arts and culture conference*. Retrieved from http://escholarship.org/uc/item/01r0k9br

Bowker, G. (2008). Software values. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Brassard, G. & Bratley, P. (1996). *Fundamentals of algorithmics*. Englewood Cliffs, NJ: Prentice-Hall.

Brooke, C.G. (2009). *Lingua fracta: Towards a rhetoric of new media*. Cresskill, NJ: Hampton Press.

Brown, G. (2012). Bug 669549 – Some DeviceManager ADB functions do not work; r=jmaher a=test-only. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/commit/6ccc1a61a0bd87fa5144aa427d697c1971b1cacd

Bruce, B.C. & Hogan, M.P. (1997). The disappearance of technology: Toward an ecological model of literacy. In D. Reinking et al. (Eds.), *Handbook of literacy and technology: Transformations in a post-typographic world* (pp. 269-281). Florence, KY: Routledge.

Burgess, H. (2010). <?php>: 'Invisible' code and the mystique of web writing. In B. Dilger & J. Rice (Eds.), *From a to <a>: Keywords of markup* (pp. 167-185). Minneapolis: U of Minnesota Press.

Burke, K. (1962). *A grammar of motives*. Berkeley and Los Angeles: U of California Press.

Burke, K. (1969). *A rhetoric of motives*. Berkeley and Los Angeles: U of California Press.

Burke, K. (1973). *The philosophy of literary form*. Berkeley and Los Angeles: U of California Press.

Carnegie, T.A.M. (2009). Interface as exordium: The rhetoric of interactivity. *Computers and Composition, 26*(3), 164-173.

Carpenter, R. (2009). Boundary negotiations: Electronic environments as interface. *Computers and Composition, 26*(3), 138-148.

Carroll, L. (1973). *Symbolic logic*. (W.W. Bartley, Ed.). New York: Clarkson N. Potter, Inc.

Cayley, J. (2002). Code is not the text (unless it is the text). *Electronic Book Review.* Retrieved from http://www.electronicbookreview.com/thread/electropoetics/literal

Ceccarelli, L. (2001). *Shaping science with rhetoric: The cases of Dobzhansky, Schrödinger, and Wilson*. Chicago: U of Chicago Press.

Chevalier, T. (1998). Work-in-progress for 'min' and 'max' properties. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/commit/a6281248abc3cb214706f970205b63ae5c83c832

Christiansen, E. (1996). Tamed by a rose: Computers as tools in human activity. In B. Nardi (Ed.), *Context and consciousness: Activity theory and human-computer interaction* (pp. 175-198). Cambridge, MA: MIT Press.

Christley, S., & Madey, G. (2007). Analysis of activity in the open source software development community. *Proceedings of the 40th Hawaii International Conference on System Sciences*. Waikoloa, HI. Retrieved from IEEE Xplore.

Chun, W.H.K. (2011). *Programmed visions: Software and memory*. Cambridge, MA: MIT Press.

Cramer, F. (2005). *Words made flesh: Code, culture, imagination*. Rotterdam: Piet Zwart Institute.

Crandall, J. (2008). Unmanned systems as assemblages. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Crowston, K., et al. (2005). A structurational perspective on leadership in free/libre open source software development teams. *Proceedings of the 1st Conference on Open Source Systems (OSS)*. Genova, Italy. Retrieved from http://floss.syr.edu

Crowston, K., & Howison, J. (2005). The social structure of free and open source software development. *First Monday, 10,* 1-27. Retrieved from http://floss.syr.edu

Crowston, K., & Howison, J. (2006). Assessing the health of open source communities. *IEEE Computer, 39,* 113-115. Retrieved from http://floss.syr.edu

Cummings, R. E. (2006). Coding with power: Toward a rhetoric of computer coding and composition. *Computers and Composition, 23*(4), 430-443.

Debian. (2011). Debian project leader. *Debian developer's corner*. Retrieved from http://www.debian.org/devel/leader.en.html

Douglass, J. (2008). #include Genre. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Douglass, J. (2011). Critical code studies conference – Week two discussion. *Electronic Book Review*. Retrieved from http://www.electronicbookreview.com/thread/firstperson/recoded

Edbauer, J. (2005). Unframing models of public distribution: From rhetorical situation to rhetorical ecologies. *Rhetoric Society Quarterly, 35*(4), 5-24.

Edmonds, J. (2008). *How to think about algorithms*. Cambridge, UK: Cambridge UP.

Emig, J. (1977). Writing as a mode of learning. *College Composition and Communication, 28*(2), 122-128.

Fagerjord, A. (2003). Rhetorical convergence: Studying web media. In G. Liestøl, A. Morrison, & T. Rasmussen (Eds.), *Digital media revisited* (pp. 293-325). Cambridge, MA: The MIT Press.

Fahnestock, J. (2011). *Rhetorical style: The uses of language in persuasion*. Oxford: Oxford UP.

Fedora. (2012). Fedora Project homepage. Retrieved from http://www.fedoraproject.org

Feenberg, A. (2002). *Transforming technology: A critical theory revisited.* Oxford: Oxford UP.

Fleckenstein, K. (2003). *Embodied literacies: Imageword and a poetics of teaching*. Carbondale: Southern Illinois UP.

Fleckenstein, K. (2005). Faceless students, virtual places: Emergence and communal accountability in online classrooms. *Computers and Composition, 22*(2), 149-176.

Ford, P. (2005). Processing Processing. In J. Spolsky (Ed.), *The best software writing I* (pp. 79-94).

França, R.M., et al. (2012). Implementing routing concerns. *GitHub*. Retrieved from https://github.com/rails/rails/commit/0dd24728a088fcb4ae616bb5d62734aca5276b1b

Fuller, M. (2003). *Behind the blip: Essays on the culture of software*. Brooklyn: Autonomedia.

Fuller, M. & Matos, S. (2011, December). Feral computing: From ubiquitous calculation to wild interactions. *The Fibreculture Journal,* 144-163. Retrieved from http://nineteen.fibreculturejournal.org/fcj-135-feral-computing-from-ubiquitous-calculation-to-wild-interactions/

Fuller, S. (1997). “Rhetoric of science”: Double the trouble? In A.G. Gross & W.M. Keith (Eds.), *Rhetorical hermeneutics: Invention and interpretation in the age of science* (pp. 279-298). Albany: SUNY Press.

Galloway, A.R. & Thacker, E. (2007). *The exploit: A theory of networks*. Minneapolis: U of Minnesota Press.

Gaonkar, D.P. (1997). The idea of rhetoric in the rhetoric of science. In A.G. Gross & W.M. Keith (Eds.), *Rhetorical hermeneutics: Invention and interpretation in the age of science* (pp. 25-85). Albany: SUNY Press.

Garsten, B. (2006). *Saving persuasion: A defense of rhetoric and judgment*. Cambridge, MA: Harvard UP.

GIMP. (2012). Getting involved. Retrieved from http://www.gimp.org/develop

GitHub. (2012). Press. *GitHub*. Retrieved from https://github.com/about/press

Gozalishvili, I., et al. (2012). Bug 7098984 – promoting SDK promise library for toolkit. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/pull/4

Grabill, J. (2003). On divides and interfaces: Access, class, and computers. *Computers and Composition, 20*(4), 455-472.

Gurak, L. (1997). *Persuasion and privacy in cyberspace*. New Haven, CT: Yale UP.

Haefner, J. (1999). The politics of the code. *Computers and Composition, 16*(3), 325-339.

Hamerly, J., Paquin, T., & Walton, S. (1999). Freeing the source: The story of Mozilla. In C. DiBona & S. Ockman (Eds.), *Open sources: Voices from the Revolution*. Retrieved April 17, 2012, from http://oreilly.com/openbook/opensources/book/netrev.html

Hayles, N.K. (2004). Print is flat, code is deep: The importance of a media-specific analysis. *Poetics Today, 25*(1), 67-90.

Hayles, N.K. (2005). *My mother was a computer: Digital subjects and literary texts*. Chicago: U of Chicago Press.

Hayles, N.K. (2012). *How we think: Digital media and contemporary technogenesis*. Chicago: U of Chicago Press.

Hillis, W.D. (1998). *The pattern on the stone: The simple ideas that make computers work.* New York: Basic.

Hocks, M.E. (2003). Understanding visual rhetoric in digital writing environments. *College Composition and Communication, 54*(4), 629-656.

Howison, J. (2006). *Coordinating and motivating open source contributors*. Retrieved from http://floss.syr.edu

Howison, J., Inoue, K., & Crowston, K. (2006). Social dynamics of free and open source team communications. *IFIP 2nd International Conference on Open Source Software.* Lake Como, Italy. Retrieved from http://floss.syr.edu

Ingo, H. (trans. 2006). *Open life: The philosophy of open source (With reader comments)*. (Trans. S. Torvalds.). Retrieved from http://openlife.cc/onlinebook

Jasinski, J. (2001). *Sourcebook on rhetoric: Key concepts in contemporary rhetorical studies*. Thousand Oaks, CA: Sage.

Jerz, D. (2007). Somewhere nearby is Colossal Cave: Examining Will Crowther's original "Adventure" in code and in Kentucky. *Digital Humanities Quarterly, 1*(2). Retrieved from http://www.digitalhumanities.org/dhq/vol/001/2/000009/000009.html

jgraham et al. (2011). GIT_WORK_TREE=. fails to apply patch in install profile when using –working-copy. *Drupal.* Retrieved from http://drupal.org/node/1276872

Johnson-Eilola, J. (2004). The database and the essay: Understanding composition as articulation. In A.F. Wysocki, J. Johnson-Eilola, C.L. Selfe, & G. Sirc (Eds.), Writing new media: Theory and applications for expanding the teaching of composition (pp. 199-235). Logan: Utah State UP.

Keith, W.M. (1997). Engineering rhetoric. In A.G. Gross & W.M. Keith (Eds.), *Rhetorical hermeneutics: Invention and interpretation in the age of science* (pp. 225-246). Albany: SUNY Press.

Kitchin, R., & Dodge, M. (2011). *Code/space: Software and everyday life*. Cambridge, MA: MIT Press.

Kittler, F. (trans. 2008). Code. In M. Fuller (Ed.), *Software studies \ A lexicon* (pp. 40-47). (Trans. T. Morrison & F. Cramer). Cambridge, MA: MIT Press.

Knuth, D. (1992). *Literate programming*. Stanford, CA: Center for the Study of Language and Information.

Kress, G. & van Leeuwen, T. (2006). *Reading images: The grammar of visual design.* 2nd ed. London and New York: Routledge.

Lakoff, G., & Johnson, M. (1980). *Metaphors we live by.* Chicago: U of Chicago Press.

Lamouri, M., et al. (2010). Bug 561636 (4/4) – When an invalid form is submitted, an error messages should be displayed. r=dolske a2.0=blocking. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/5d30f398bd39d63e9938165f9def84e2218c8589/browser/base/content/browser.js

Lanham, R.A. (1993). *The electronic word: Democracy, technology, and the arts.* Chicago: U of Chicago Press.

Lanham, R.A. (2003). *Analyzing prose.* 2nd ed. New York: Continuum.

Lawson, K. (2011). Remembering Diaspora*: The open source social network. *The Chronicle of Higher Education*. Retrieved from https://chronicle.com/blogs/profhacker/remembering-diaspora-the-open-source-social-network

Le Lionnais, F. (2007). Lipo: First manifesto. In W. Motte (Ed.), *Oulipo: A primer of potential literature* (pp. 26-28). Champaign, IL: Dalkey Archive Press.

Lunenfeld, P. (2008). Counterprogramming. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

MacCormick, J. (2011). *Nine algorithms that changed the future: The ingenious ideas that drive today's computers*. Princeton, NJ: Princeton UP.

Manovich, L. (2001). *The language of new media*. Cambridge, MA: MIT Press.

Manovich, L. (2008a). Cultural analytics. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Manovich, L. (2008b). *Software takes command*. Retrieved from http://www.softwarestudies.com/softbook/manovich_softbook_11_20_2008.pdf

Manovich, L. (2011). Mondrian vs Rothko: Footprints and evolution in style space. *Software Studies*. Retrieved from http://lab.softwarestudies.com/2011/06/mondrian-vs-rothko-footprints-and.html

Marino, M. (2006). Critical code studies. *Electronic Book Review*. Retrieved from http://www.electronicbookreview.com/thread/electropoetics/codology

Marino, M. (2008). Critical code studies. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Mateas, M. (2005). Procedural literacy: Educating the new media practitioner. *On the Horizon, 13*(2), 101-111.

Matsumoto, Y. (2007). Treating code as an essay. In A. Oram & G. Wilson (Eds.), *Beautiful code: Leading programmers explain how they think* (pp. 477-481). Sebastopol, CA: O'Reilly.

McCorkle, B. (2012). *Rhetorical delivery as technological discourse*. Carbondale and Edwardsville: Southern Illinois UP.

McKeon, R. (1971). The uses of rhetoric in a technological age: Architectonic productive arts. In L.F. Bitzer & E. Black (Eds.), *The prospect of rhetoric* (pp. 44-63). Englewood Cliffs, NJ: Prentice Hall.

McNenny, G., & Roen, D. (1992). The case for collaborative scholarship in rhetoric and composition. *Rhetoric Review, 10*(2), 291-310.

Miller, C.R. (1994). Opportunity, opportunism, and progress: *Kairos* in the rhetoric of technology. *Argumentation, 8*(1), 81-96.

Miller, C.R. (2000). The Aristotelian *topos*: Hunting for novelty. In A. Gross & A. Walzer (Eds.), *Rereading Aristotle's* Rhetoric. Carbondale: Southern Illinois UP.

Miller, C.R. (2001). Writing in a culture of simulation: Ethos online. In Coppock, P. (Ed.), *The semiotics of writing: Transdisciplinary perspectives on the technology of writing* (pp. 253–279). Turnhout, Belgium: Brepols Publishers.

Miller, C.R. (2007). What can automation tell us about agency? *Rhetoric Society Quarterly, 37*(2), 137-157.

Miller, C.R. (2010a). Foreword: Rhetoric, technology, and the pushmi-pullyu. In S.A. Selber (Ed.), *Rhetorics and technologies: New directions in writing and communication* (pp. ix-xii). Columbia: U of South Carolina Press.

Miller, C.R. (2010b). Should we name the tools? Concealing and revealing the art of rhetoric. In J.M. Ackerman & D.J. Coogan (Eds.), *The public work of rhetoric: Citizen-scholars and civic engagement* (pp. 19-38). Columbia: University of South Carolina Press.

Miller, C.R. (2012, July). [Interview with M.A. Mascarenhas]. *Figure/Ground Communication.* Retrieved from http://figureground.ca/interviews/carolyn-r-miller/

Montfort, N. & Bogost, I. (2009). *Racing the beam: The Atari video computer system.* Cambridge, MA: MIT Press.

Moody, G. (2012). Interview: Linus Torvalds—I don't read code any more. *The H open.* Retrieved from http://www.h-online.com/open/features/Interview-Linus-Torvalds-I-don-t-read-code-any-more-1748462.html

Moretti, F. (2005). *Graphs, maps, trees: Abstract models for literary history.* London: Verso.

Mosedale, D., & Vande Beken, P. (2002). Initial work (mostly by peterv) for a bayesian spam filter. Not yet part of the build. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/b7890a4bbb26100ef95f8872b10e18349d961400/mailnews/extensions/bayesian-spam-filter/menuOverlay.js

Moxley, J. (2008). Datagogies, writing spaces, and the age of peer production. *Computers and Composition, 25*(2), 182-202.

Mozilla. (n.d.a). Governance. *Mozilla*. Retrieved from http://www.mozilla.org/about/governance.html

Mozilla. (n.d.b) Mozilla at a glance. *Mozilla*. Retrieved from http://www.mozilla.org/en-US/press/ataglance

Mozilla. (2012a). Creating Mercurial user repositories. *Mozilla Developer Network.* Retrieved from https://developer.mozilla.org/en-US/docs/Creating_Mercurial_User_Repositories

Mozilla. (2012b). Mozilla. *GitHub*. Retrieved April 17, 2012, from https://github.com/mozilla

Mozilla. (2012c). Tinderbox. *Mozilla Developer Network*. Retrieved from https://developer.mozilla.org/en/Tinderbox

Muckelbauer, J. (2008). *The future of invention: Rhetoric, postmodernism, and the problem of change*. Albany: SUNY Press.

Murray, J. (2009). *Non-discursive rhetoric: Image and affect in multimodal composition*. Albany: SUNY Press.

Nakakoji, K., Yamada, K., & Giaccardi, E. (2005). Understanding the nature of collaboration in open-source software development. *Proceedings of the 12th Asia-Pacific software engineering conference*. Taipei, Taiwan. Retrieved from IEEE Xplore.

Nardi, B.A. (1995). *A small matter of programming: Perspectives on end user computing*. Cambridge: MIT Press.

Netcraft. (2012). November 2012 Web Survey. Retrieved from http://news.netcraft.com/archives/2012/11/01/november-2012-web-server-survey.html

Nutter, C.O., et al. (2012). Incorporate OpenSSL tests from JRuby. *GitHub*. Retrieved from https://github.com/ruby/ruby/pull/206

O'Reilly, T. (2005). The open source paradigm shift. In C. DiBona, D. Cooper, & M. Stone (Eds.), *Open sources 2.0: The continuing evolution* (pp. 253-272). Sebastopol, CA: O'Reilly Media.

Parikka, J. (2008). Copy. In M. Fuller (Ed.), *Software studies \ A lexicon.* Cambridge, MA: MIT Press.

Payne, D. (2005). English studies in Levittown: Rhetorics of space and technology in course-management software. *College English, 67*(5), 483-507.

Perelman, C. & Olbrechts-Tyteca, L. (1969). *The new rhetoric: A treatise on argumentation*. (Trans. J. Wilkenson & P. Weaver). Notre Dame, IN: U of Notre Dame Press.

Platt, D.S. (2007). *Why software sucks ... and what you can do about it.* Upper Saddle River, NJ: Addison-Wesley.

Porter, J. (2009). Recovering delivery for digital rhetoric. *Computers and Composition, 26*(4), 207-224.

Prelli, L.J. (1989). The rhetorical construction of scientific ethos. In H.W. Simmons (Ed.), *Rhetoric in the human sciences: Inquiries in social construction* (pp. 48-68). London: Sage.

Queneau, R. (2007). Potential literature. In W. Motte (Ed.), *Oulipo: A primer of potential literature* (pp. 51-64). Champaign, IL: Dalkey Archive Press.

Quine. (n.d.) In *Rosetta code*. Retrieved from http://rosettacode.org/wiki/Quine

Quine, W.V. (1976). The ways of paradox. In W.V. Quine, *The ways of paradox and other essays*. Harvard: Harvard UP. Retrieved from http://www.math.dartmouth.edu/~matc/Readers/HowManyAngels/Paradox.html

Raley, R. (2002). Interferences: [Net.Writing] and the practice of codework. *Electronic Book Review*. Retrieved from http://www.electronicbookreview.com/thread/electropoetics/net.writing

Ramsay, S. (2011). *Reading machines: Toward an algorithmic criticism*. Urbana, Chicago, and Springfield: U of Illinois Press.

Raymond, E. S. (2000). *The cathedral and the bazaar*. Retrieved from http://www.catb.org/~esr/writings/homestead-ing/cathedral-bazaar/index.html

Rieder, D. (2010). Snowballs and other numerate acts of textuality: Exploring the 'alphanumeric' dimensions of (visual) rhetoric and writing with ActionScript 3. *Computers and Composition Online*. Retrieved from http://www.bgsu.edu/cconline/rieder/

Sack, W. (2008). From software studies to software design. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Sample, M. (2011). Platform studies as historical inquiry, or, videogames bleed history. *Play the past*. Retrieved from http://www.playthepast.org/?p=1857

Scott, Z., & Bosslet, M. (2012). Feature #7400: Incorporate OpenSSL tests from JRuby. *Ruby Issue Tracking System*. Retrieved from http://bugs.ruby-lang.org/issues/7400

Selfe, C.L. & Selfe, R.J., Jr. (1994). The politics of the interface: Power and its exercise in electronic contact zones. *College Composition and Communication, 45*(4), 480-504.

Shannon, C.E. (1937). *A symbolic analysis of relay and switching circuits.* (Unpublished master's thesis). Massachusetts Institute of Technology, Cambridge, MA.

Sharp, G., et al. (2010). Bug 595356: pref dialog should show default home page as “Firefox start” .r=mak a=blocking. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/443b32151a08f0ad62d39291b4ad5e7d5b65521e/browser/components/preferences/main.js

Sherov, M., et al. (2012). Fixes #12587, 'hidden' selector doesn't work for SVG images in Firefox. *GitHub*. Retrieved from https://github.com/jquery/jquery/pull/939

Shiffman, D. (2011). HashMap example. *Processing*. Retrieved from https://code.google.com/p/processing/source/browse/trunk/processing/java/examples/Topics/Advanced+Data/HashMapClass/

Shipka, J. (2005). A multimodal task-based framework for composing. *College Composition and Communication, 57*(2), 277-306.

Shipka, J. (2011). *Towards a composition made whole*. Pittsburgh: University of Pittsburgh.

Shirky, C. (2009). *Here comes everybody: The power of organizing without organizations*. New York: Penguin.

Silver, D. (2005). Selling cyberspace: Constructing and deconstructing the rhetoric of community. *Southern Communication Journal, 70*(3), 187-199.

Sondheim, A. (2001). Introduction: Codework. *American Book Review, 22*(6), 1-4.

Sorapure, M. (2010). Information visualization, Web 2.0, and the teaching of writing. *Computers and Composition, 27*(1), 59-70.

Steiner, C. (2012). *Automate this: How algorithms came to rule the world*. New York: Penguin.

Stewart, D. (2005). Social status in an open-source community. *American Sociological Review, 70*(5), 823-842.

Terdiman, D. (2008). Taking on Twitter with open-source software. *CNET News*. Retrieved from http://news.cnet.com/8301-13772_3-10058946-52.html

Thompson, K. (1984). Reflections on trusting trust. *Communications of the ACM, 27*(8), 761-763.

Trasviña, F., et al. (2012). Introduction to Object-Oriented Javascript. *Mozilla Developer Network*. Retrieved from https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript

Turkle, S. (1984). *The second self: Computers and the human spirit*. New York: Simon & Schuster.

Vatz, R.E. (1968). The myth of the rhetorical situation. *Philosophy & Rhetoric, 6*(3), 154-161.

Vee, A. (2010). *Proceduracy: Computer code writing in the continuum of literacy*. Diss. University of Wisconsin-Madison.

Voswinkel, L. (2012). Pull request #554: Fixed a small, but major, typo. *GitHub.* Retrieved from https://github.com/hotsh/rstat.us/pull/554

Walden, J. et al. (2006a). Add a few missed files...if I have a clean tree and patch it with a patch that includes files, I shouldn't have to cvs add those files before I commit them – cvs sucks. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/aac4ed2044d52f80cf9e597f86f7c37c747720c8/browser/components/preferences/main.js

Walden, J. et al. (2006b). Bug 340677 – update preference panels (add anti-phishing, rationalize categories, simplify wording). We're slowly spiralling in on a final design... r=mconnor. *GitHub*. Retrieved from https://github.com/mozilla/mozilla-central/blob/aac4ed2044d52f80cf9e597f86f7c37c747720c8/browser/components/preferences/main.js

Walker, J. (1994). The body of persuasion: A theory of the enthymeme. *College English, 56*(1), 46-65.

Wall, L. (1990). Black Perl. *Best of Internet*. Retrieved from http://internet.ls-la.net/comppoems/black-perl.html

Walton, D. (2001). Enthymemes, common knowledge, and plausible inference. *Philosophy and Rhetoric, 34*(2), 93-112.

Wardrip-Fruin, N. (2008). Expressive processing. *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Wardrip-Fruin, N. (2009). *Expressive processing: Digital fictions, computer games, and software studies*. Cambridge, MA: MIT Press.

Warnick, B. (2002). *Critical literacy in a digital era: Technology, rhetoric, and the public interest.* Mahwah, NJ: Lawrence Erlbaum Associates.

Warnick, B. (2004). Online ethos: Source credibility in an "authorless" environment. *American Behavioral Scientist, 48*(2), 256-265.

Warnick, B. (2007). *Rhetoric online: Persuasion and politics on the World Wide Web*. New York: Peter Lang.

Wiggins, A., Howison, J., & Crowston, K. (2008). Social dynamics of FLOSS team communication across channels. *Proceedings of the IFIP 2.13 Working Conference on Open Source Software (OSS)*. Milan, Italy. pp. 131-142. Retrieved from http://floss.syr.edu

Wysocki, A.F. (2005). awaywithwords: On the possibilities in unavailable designs. *Computers and Composition, 22*(1), 55-62.

W3Counter. (2012). Web browser market share trends. *W3Counter*. Retrieved from http://www.w3counter.com/trends

Yancey, K.B. (2004). Made not only in words: Composition in a new key. *College Composition and Communication, 56*(2), 297-328.

Zappen, J. (2005). Digital rhetoric: Toward an integrated theory. *Technical Communication Quarterly, 14*(3), 319-325.

Zhigunov, R., et al. (2012). Add support for AR5BBU22 [0489:e03c]. *GitHub*. Retrieved from https://github.com/torvalds/linux/pull/17