CHAPTER 1

Toward the Rhetorical Study of Code

In April 2014, it was revealed that a security bug in OpenSSL, a software library for ensuring secure communication in and across computer networks, had existed for the previous two years. The bug operated by opening an exploit into the "heartbeat request message," a means of testing the security of connections opened with OpenSSL. While these messages were *supposed* to send a specific kind of data (a 16-bit integer) for this test and then have the same message sent back to confirm connection, the bug allowed for the data contents of a computer's entire allocated memory buffer to be sent as part of the message. Dubbed "Heartbleed," the bug risked transmission of incredibly sensitive data by accident or by malicious intent, as some attackers could send heartbeat requests specifically to receive their targets' memory buffer data. Because OpenSSL is a popular and widely used library, the security impact of Heartbleed included such varied systems as Amazon Web Services, Minecraft, Reddit, SoundCloud, Steam, Stripe, Tumblr, and Wikipedia.

The code responsible for the bug initially set up a particular way of reading, storing, and sending relevant information. This way just so happened to compromise the central goals of OpenSSL by allowing a requesting agent to seek out memory buffer data from unsuspecting or otherwise vulnerable targets, and these requests could be repeated infinitely (Cassidy 2014). While the exact nature of any collected data could never be precisely determined before a given request, the likelihood of *something* valuable being collected could be all but guaranteed with continued exploitation of the bug.

The eventual patch for the bug, accepted for distribution not long after the bug was revealed, was relatively small in scope; an excerpt of the patch is shown in table 1.1. The code selected performs a relatively simple task: it ensures that no zero-length heartbeat requests are sent, and then it ensures that the length of a given heartbeat (the "payload") has an appropriately long record; otherwise, the request is discarded (Cassidy 2014). Despite the relative simplicity of the code in terms of the overall complexity and number of lines of code involved, its significance was enormous, with only 12,000 of the 800,000 most popular websites remaining vulnerable more than a month after the bug was made public (Leyden 2014).

So, given the potential consequences of Heartbleed and the relatively small fix it took to resolve the issue, why did it persist for so long? And what does the story tell us about the rhetorical activity taking place not only in response to the bug, of which there are many examples to choose from, but also *within* the code texts before and after its patch? Further, to what extent does Heartbleed—or any other specific example, such as the case of the Mozilla web browser, discussed later in this book—reflect more broadly applicable understanding in regards to software development, use, and the ways programmers approach their code as meaningful communication?

These questions cannot all be answered easily. My goal, however, is not to propose easy answers so much as to attend to the tensions, problems, and complications that arise from circumstances like those of the Heartbleed bug. Further, I hope to draw more critical attention to the rhetorical activity taking place in and through the code at the center of phenomena like the Heartbleed bug, as that activity *can* and *does* exert tremendous influence over how individuals and communities respond to and deal with the consequences of writing code—and thus constructing and disseminating meaning in particular ways for certain audiences. This attention takes place through the cultivation of a rhetorical orientation toward software, code, and its algorithmic procedures, which we can recognize through scholarly literature in several fields: digital rhetoric, software studies, critical code studies, and technical communication.

Table 1.1. Excerpt from Heartbleed patch (t1_lib.c) by snhenson et al. (2015)

Line	Code
3977	`/* Read type and payload length first */`
3978	`if (1 + 2 + 16 > s->s3->rrec.length)`
3979	`        return 0; /* silently discard */`
3980	`hbtype = *p++;`
3981	`n2s(p, payload);`
3982	`if (1 + 2 + payload + 16 > s->s3->rrec.length)`
3983	`        return 0; /* silently discard per RFC 6520 sec. 4 */`
3984	`pl = p;`

Over the past decade, there has been continually increasing interest in the rhetorical study of software and code, with examinations of software as possessing and communicating ethics (Brown 2015) to advocacy for code-related literacy (Vee 2017) to an understanding of video games as a form of embodied practice (Holmes 2017) to the potential emergence of genres in code texts (Brock and Mehlenbacher 2017), as well as to experimentation with writing meant for algorithmic readers (Gallagher 2017). This interest, as well as its broadening scope and deepening focus, suggests the birth of a scholarly field related to but not incorporated entirely within rhetoric nor within the fields of software studies or critical code studies. Instead, this field, which I call "rhetorical code studies" (a term that has already seen some uptake; see Beck 2016), exists at a point of convergence of all three areas of study; its name reflects the position and focus of inquiry that the field has developed thus far and the possibilities that future investigations might uncover. Rhetorical code studies provides a means by which software use and development as well as the communicative work that takes place through its code texts—as well as the algorithmic logics communicated through those code texts (Beck 2016)—could be understood more clearly as rhetorical activity. Further, such inquiry could open up new directions for pedagogical engagement with code and computation as avenues for communication as well as for critical literacy.

An argument for the rhetorical study of code might seem either as obvious as the rhetorical study of other forms of communication or as preposterous as the rhetorical study of an arhetorical subject, and this reflects two common threads of conversation among rhetoricians regarding the scope of the discipline. It is neither compelling nor accurate to argue that rhetoricians should attend to code simply because such inquiry has not been performed much or at all. Instead, I base my argument on the premises that code is as meaningful as any other form of communication and that the sheer amount of code produced each day (likely numbering in the billions of lines of code), along with the impact that much of that code has on myriad aspects of daily life, suggests an important phenomenon in need of continued and focused investigation.

Rhetorical code studies can potentially address multiple kinds and modes of rhetorical activity and interaction, including communication reflecting complex relationships such as among developers, between developers and users, between developers and technologies, and also users and technologies. For the purposes of this project, and explained in more detail in chapters 2 and 3, I emphasize the rhetorical activity of developer-to-developer communication, as that sort of interaction has significant impact on subsequently triggered sites of activity, such as in how technological interfaces communicate particular arguments to their users.

What Does Rhetorical Code Studies Involve?

Rhetorical code studies, as I am arguing for it to be described and understood, combines the study of rhetoric with that of software and code, possessing a particular focus of inquiry centered on several key characteristics of code and the discursive contexts surrounding its composition. This description of rhetorical code studies builds on Beck (2016), who argued for an understanding of algorithms as being fundamentally persuasive and advocated for the rhetorical study thereof; code serves as an optimal site for exploring how algorithmic expression and communication operates rhetorically, thanks to a complex set of relationships between code, language, software, and procedure.

The first characteristic of code important to its rhetorical study is the set of rhetorical qualities and capacities of code, a term that refers to both source code and the executable programs built from source code. Source code is the readable set of operational commands written in any number of existing computer languages, generally created and stored as text files with language-specific syntax and vocabulary. 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 (translated) into a lower-level language in order to be executed by a computer. High-level languages include C, Java, and Ruby; these tend to possess features that more closely approach those of natural languages like English. Low-level languages include Assembly and other forms of machine code, and they specify far less human-readable instructions to the computer(s) executing their commands. Executable programs, in contrast, are those compiled or otherwise machine-interpreted software that allow a user or machine to perform particular activities, such as browsing the web, writing and saving text documents, calculating mathematical equations, verifying password correctness, and so on.

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 code texts via statements called *comments*, which are distinguished from code statements in that comments are noninterpretable, nonexecuting natural language text blocks intended for human audiences. Because of their existence within code files and their proximity to particular lines of code, comments tend to illuminate some intended purpose for or function of a given block of code. (Of course, sometimes this illumination doesn't happen, and the result is more confusing than clarifying.) An example of comments distributed amid code statements can be seen in the Heartbleed patch excerpt demonstrated in table 1. Lines 3977, 3979, and 3983 each have brief phrases directed toward human readers and punctuated in a way as to let the computer know to ignore those phrases, e.g. from line 3979: "**`/* silently discard */`**" (snhenson et al. 2015). In this comment, the author indicates to interested readers what the code in that line *should* successfully perform; if some other behavior occurs, then contributors to the program can more easily identify the likely source of the problem. Comments exist among but outside the functional scope of the source code lines that make up the interpretable or compileable software program. In addition to intracode 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) "among" code operations as well as within their logics.

A third characteristic of code is the set of social, cultural, and historical contexts that facilitate its composition, dissemination, and critique in general as well as in regards to specific programs or contexts for use of technologies. While these contexts are connected to the rhetorical qualities of code broadly speaking, it is important to distinguish them here because these issues are discussed frequently and prominently in a number of public and academic circles. Some of this work includes examining issues of race and relevant access, or lack of access, to technologies in the classroom and beyond (Banks 2006). Nakamura (2007) illuminated the tensions between how people of color are frequently depicted via digital media and how they represent themselves and develop online communities. McPherson (2012) and Risam (2015) have discussed the problematic relationship between digital technologies, their histories, and the racial makeup of the academic communities that study them, especially in the conglomeration of fields called the "digital humanities" and how it positions inquiry in relation to its objects of study. Other scholars have attended to the historical trajectory of computer technologies, the gendering of labor relating to their production and innovation, and the social and political impacts of that gendering (Abbate 2012; Hicks 2017). Still others, like Coleman (2013) and Kelty (2008), have examined cultural concerns and contexts surrounding software use, especially free and open source software and its various development, user, and hacker communities.

These characteristics of code—its rhetorical qualities, the discourse surrounding its development, and the social and cultural contexts in which it is composed and employed—inform the premises on which rhetorical code studies is built. In addition to providing an avenue for focused critical inquiry of underexplored objects of study and perspectives for study regarding technological development and use, rhetorical code studies would offer new means of engaging in the creative and rhetorically informed *production* of meaningful code and software. Specifically, I posit that software that has been composed with as much attention to the rhetorical dimensions of its code as to its intended and anticipated execution and output—as opposed to code perceived to be primarily or wholly instrumental in nature—might well herald valuable innovations for approaching computer use in general and code composition in particular.

I hope to describe rhetorical code studies and how its object might be more fully scrutinized precisely because the use and development of software and code can help us understand more clearly, across a number of rhetorical dimensions, how we attempt to communicate meaningfully with others across digital contexts. For rhetoric in particular, a field in which scholars seek out knowledge of and proficiency with meaningful expression relating to a given situation, this help is key. 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 making meaning.

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, concepts that have had longstanding importance to rhetoric. The first of these is *action* as understood through Burke's (1969) term *symbolic action*. For Burke, symbolic action refers to any communicative effort the meaning of which extends beyond the specific set of acts that makes up the transmission of a particular message. The second concept central to this argument is *meaning making*, the efforts by one or more rhetors to induce or influence an audience to act in response to a particular message, whether that action is physical or not. The third and final major concept 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" (147). 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. Following that analysis, I offer another means of rhetorical inquiry into code by working through a series of exercises intended to highlight concepts important to rhetoric and to programming so that an interested reader—even one not necessarily familiar with software development—can practice rhetorical invention through the composition of code.

With these ideas in mind, rhetoricians and scholars of code alike could move forward along a number of trajectories that would provide insight into the meaningful, rhetorical, and critical values of computational objects of study. To better understand how we might move forward in these ways, however, it is first necessary to gain a sense of how these fields have set the stage for rhetorical code studies to blossom and for these trajectories to become available for us to pursue.

Digital Rhetoric

While rhetoric as a discipline is not inherently focused on digital media, it has nonetheless incorporated digital forms of meaning making into its fold, with scholars revising and inventing theories of rhetoric as necessary to more fully understand how those forms operate rhetorically. Below, I outline the relationship between digital rhetoric, its subfield of computers and writing, and the emergence of procedural rhetoric as a lens through which to understand how code, through algorithmic procedure, constructs and communicates meaning.

Rhetoric—and most notably rhetoric and composition—has expressly held an interest in digital technologies for communicative ends at least since the birth of the Computers and Writing conference in 1982, when a number of scholars and consultants gathered to discuss "the place of computing in the writing curriculum" (Gerrard 1995, 280). While the primary focus for the conference's first several meetings was on the facilitation of student writing with computer technologies and the analysis of that writing, discussions over the next several meetings of the conference quickly expanded to include the social and political impacts of those technologies and the development of theories relating to computer-based writing activities in and out of the classroom.

During this time, Lanham (1993) coined the term “digital rhetoric” to describe a number of possibilities for critical and practical engagement via computer-based communication. For Lanham and many scholars since, the “bi-stable oscillation” between looking *at* and *through* texts—that is, paying attention to and looking beyond the qualities of a given medium (Lanham 1993, 5)—remains a central component of the act of reading and composing digital texts through continually emerging means.

In the twenty-five years since the birth of “digital rhetoric,” rhetoricians have explored myriad dimensions of emerging technologies for expression, persuasion, and other forms of meaning making, as well as how writer, technology, community, and society interrelate in complex ways, ways that extend from existing media and modes of communication and that are unique to digital media. Because of the potential ambiguity of the term “digital rhetoric,” some scholars have attempted to define more clearly its ever-widening scope. In particular, Losh (2009) outlined four distinct, although related, definitions that reflected major threads of relevant critical investigation:

1. The conventions of new digital genres that are used for every discourse, as well as for special occasions, in average people’s lives.
2. Public rhetoric, in the form of political messages from government institutions, which is represented or recorded through digital technology and disseminated via electronic distributed networks.
3. The emerging scholarly discipline concerned with the rhetorical interpretation of computer-generated media as objects of study.
4. Mathematical theories of communication from the field of information science, many of which attempt to quantify the amount of uncertainty in a given linguistic exchange or the likely paths through which messages travel. (47–48)

Losh’s definitions clearly include not only humanistic inquiry but the broader employment of rhetoric as constructing meaning for diverse purposes. Her fourth definition is perhaps most significant for this project, as the valuable and relevant work occurring in information science identified by Losh informs much of the philosophy behind rhetorical code studies.

Eyman (2015) has offered a similar broadened set of definitions of “digital rhetoric” (although focused primarily on scholarly contexts), spanning from classical rhetoric to digital media studies to computers and writing. Eyman argued, through his synthesis of the numerous articulations of the term he provided, that rhetoric had much to offer and gain from the diversity of approaches being undertaken throughout academe, only some of which explicitly employ the term "digital rhetoric." While much of the work in these approaches has focused on several specific spheres of communication (namely civic, academic, and professional), there has been a growing amount of scholarship expanding more broadly and deeply our understanding of how digital technology facilitates rhetorical activity across nearly all aspects of our lives.

Among the most significant arguments about this facilitation is provided by Selfe and Selfe (1994), who 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 to identify anticipated user populations of those interfaces and what sorts of knowledge, technical proficiency, or other forms of awareness the users were expected to possess for the interfaces' use (Selfe and Selfe 1994). Grabill (2003) articulated the difficulty many rhetoricians had 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. (465–66)

In essence, rhetoricians have historically struggled with overtly technological concerns simply because those concerns are frequently 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 constructing and expressing meaning.

Writing (With) Digital Media

Rhetoricians have pursued several equally important tracks of inquiry regarding digital media and how it relates to existing paradigms of rhetorical theory. One group of rhetorical critics has explored how historical approaches to rhetoric could effectively provide insight to communication taking place through electronic and digital media. Miller (2007) examined the concepts of *ethopoeia* and rhetorical agency in relation to automation and similar complex technological engagements, using the example of an automated software program for writing assessment. Warnick (2007) argued that rhetoric as a discipline must move away from notions of single authors and works as the bases for its study and instead turn toward the distinct concepts of collaborative authorships and interlinked hypertexts (122). Brooke (2009) 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. Carnegie (2009) examined how technological interfaces might be better understood as contemporary forms of *exordium*; Tarsa (2015) expanded on this argument to explore how students make sense of digital interface through that lens, especially in regards to interfaces that promote participation and interactivity.

A second group of rhetorical scholars has worked to expand critical orientation and language so as to more fully explain and contextualize discursive practices mediated by digital technologies. For example, Fagerjord (2003) argued that rhetoric as a term needed to be redefined to incorporate more diverse kinds and dimensions of emerging media. Zappen (2005) posited that it is necessary 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). Porter (2009) examined delivery as a means of revising existing rhetorical theory or producing new theoretical perspective: "[t]he real value in developing a robust rhetorical theory for digital delivery lies in [productive action]" (221). Losh (2016) argued for the need to attend to rhetorical situation, exigence, orientation, and navigation in regards to "new forms of rhetorical performance by computational components [that] may be going on independent of human-centered display" and not merely those technological engagements easily accessible or recognizable to us (n.p.). Meanwhile, Stolley (2014) drew an explicit connection between conventional and digital approaches to meaning making in order to advocate for a transformation of rhetorical thought, arguing outright that "programming is writing. I mean that literally" (264).

From Digital to Procedural

Scholars interested in computers and writing have made 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 there might be merit 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. Shepherd (2016) examined how protocological systems—in particular, online matching technologies—influence particular habits and behaviors. 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 (48).

More recently, special issues of two journals—*enculturation*, edited by Hodgson and Barnett (2016), and *Computational Culture*, edited by Brown and Vee (2016)—focused on new directions and perspectives considered by rhetoricians in regard to meaningful communication in and through technology, computation, and procedure. Among the authors in the special issue of *enculturation*, Holmes (2016) explored how technologies work persuasively to effect behavioral change in users, connecting behavioral habits (*hexeis*) with *ethos* to describe how particular digital practices occur. Beck (2016), explicitly responding in part to an earlier call for the rhetorical study of code, argued for a need by rhetoricians to attend more fully and closely to computer algorithms as rhetorical agents, outlining how they can be understood to function persuasively. Rieder (2016) demonstrated the possibilities for hybrid digital-physical "eversions" that made sociopolitical interventions via rhetorical engagements with and uses of digital data, such as the depth data generated by a Microsoft Kinect sensor. Juszkiewicz and Warfel (2016) examined the rhetorical nature of mathematics, especially in how mathematics programming affords and constrains particular kinds of meaning making. Some of the authors included in the special issue of *Computational Culture* include Brock (2016), who drew attention to the relationship between rhetorical style and the composition of code; similarly, Bellinger (2016) complicated the ways in which digital media scholars have identified error, failure, and disruption in contrast to successful or "proper" function. Birkbak and Carlsen (2016) offered Facebook's EdgeRank algorithm as a demonstration of procedural rhetoric in that the algorithm itself becomes "a rhetor that actively constructs a rhetorical commonplace that can be drawn upon in order to justify" its own procedural expression (n.p.). Maher (2016) focused on *phronesis* in regards to how "artificial rhetorical agents" are being developed to consider complex and nuanced ethical situations.

Perhaps the most influential scholar for the combined study of rhetoric and code is Bogost (2007), whose concept of *procedural rhetoric* has articulated a means of inducing action as demonstrated through particular media in order to teach audiences how to *use* those media; this concept has been expanded on and connected more fully to traditions of rhetoric by Ingraham (2014) and Beck (2016), among others. Procedural rhetoric, according to Bogost, is "the practice of using processes persuasively" (28). This definition is most commonly considered in regards to the expressive outcomes of processes (e.g., software interfaces), but procedural construction is a form of meaning making as well: it sets up *how* algorithmic processes can be, and are, employed rhetorically. 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" (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 induced are drastically altered by user and computer system alike.

Rhetoricians are clearly interested and involved in work relating to how meaningful communication occurs in digital contexts, from extensions of traditional or conventional forums and channels to emerging genres and situations, including a strong focus on algorithmic procedures as underlying logics for digital media. Rhetorical code studies can build on this work by providing continued and heightened attention to how the technologies underlying those contexts enable, augment, and constrain the construction of meaning with and through digital technologies.

Critical Code Studies

A related body of scholarly inquiry connecting software and rhetoric emerges primarily from the study of literature, in which critical methodologies are applied to code as if it possessed familiar textual qualities and functioned similarly to literary texts. Scholars interested in critical code studies explore how particular code languages facilitate certain habits of mind as well as means of communication via the construction of software programs. This line of inquiry sprung from early engagements with software studies and new media scholarship as part of a directed effort to explore computer code as something "more" (i.e., more meaningful and significant) than its output (Cayley 2002; Marino 2006).

Many critics of code approach the field with a literary orientation, viewing code texts as insightful regarding questions of authorial intent, contemporary trends in writing (programming) style and genre, and meaning as expressed within a particular body of code (Marino 2006). Ramsay (2011) made use of code for such ends, performing an "algorithmic criticism" that enabled him to transform or "deform" texts into code-mediated expressions (referred to as "paratexts") so as to highlight new avenues for textual interpretation. For Ramsay, algorithmic code serves as an innovative means for scholarly engagement because of its ability to make familiar texts strange so as to help critics generate new questions and avenues for critical research. Douglass (2011) has approached code more traditionally as a textual object for inquiry, questioning how code *is currently* read as distinct from and counter to how code *should be* read. That is, rather than suggest "best practices" for code-related interpretation, Douglass has emphasized how various groups tend to evaluate and value code as a familiar or unfamiliar form of meaningful communication; subsequently, Douglass has suggested, we need to ask what these existing practices might mean for our ability to work with code in new and potentially significant ways.

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 the user looks 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 in order to understand how specific lines of code in its files communicated significant meaning to the game player about how to explore certain possibilities for game play. Sullivan (2013) developed a conference presentation webtext whose full content would only reveal itself when a reader examined its source code, where the bulk of Sullivan's argument existed as comments hidden from web browser rendering. Schlesinger (2013) explored the possibility of a "feminist programming language," one that might not be based on Boolean logic, in a HASTAC blog post that garnered considerable discussion regarding its potential impact on programming philosophy (namely, about the extent to which existing technological constraints would allow such a language to function). Schlesinger's work is also noteworthy for the nearly immediate backlash it received from some programmer circles whose members deemed its theoretical language "academic gibberish" and who offered thinly veiled and overt misogynistic criticisms at Schlesinger and other discussion participants (cf. topwiz 2013; Reif 2013).

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. In contrast, Hayles (2005) argued for a distinction between natural language and executable code language that emphasized the ability of natural language communication to signify more than it literally suggests, subordinating code to mere description of 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.

A third group combines the study of code and algorithmic procedure with that of literacy; its scholars have called attention to the growing public conversation surrounding "computational literacy" and "coding literacy" (Vee 2017), "procedural literacy" (Mateas 2005), and even "iteracy" (Berry 2011) as concepts and means of promoting STEM-related education and vocational training. These calls resemble earlier arguments in technical communication (e.g. Miller 1979) for technical documents to be viewed as rhetorical rather than merely instrumental and that writing students cultivate critical and rhetorical literacies regarding technologies in addition to becoming technically proficient in using them (Selber 2004). Examinations of these more recent calls for and approaches to promoting various forms of computational literacy demonstrate that the rhetorical dimensions of code and coding are frequently left implicit in these literacy-focused settings even while a number of creative problem skills are promoted through programming activities (Vee 2017, 16). The result of this apparent imbalance in critical awareness is a proliferation of lay arguments that advocate improving critical thinking (as provided within the bounds of STEM initiatives) but that also discount or overlook the contributions that humanities fields could make to help promote the very sort of critical thinking desired by educators and employers. A focused effort to tie together explicitly the goals of the computational literacy movement with the skills and knowledge of critical analysis and practice in the humanities—that is, an effort very much in the wheelhouse of rhetorical code studies and which is already being championed by Vee, Berry, and others—could lead to a much stronger and effective set of literacy initiatives.

Software Studies

The field of software studies, while closely related to the critical study and analysis of code, 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, concerns of clear relevance to rhetorical code studies. There are several major initiatives currently being developed by software scholars beyond the set of varied approaches to the study of software by critics who affiliate themselves, or who are otherwise associated, with the field.

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 (2008) defined the goal of software studies as "investigat[ing] both the role of software in forming contemporary culture, and cultural, social, and economic forces that are shaping development of software itself" (5). A number of disciplines are represented in software studies, reflecting the broad range of its subject's impact, from art (Crandall 2008) and design (Lunenfeld 2008; Sack 2008) to science and technology studies (Bowker 2008) and literary studies (Douglass 2008).

Platform studies deserves a brief note as a field closely related to software studies due to its focus on the ecologies of software and hardware technologies that 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 (see Bogost 2008; Montfort and Bogost 2009), although the field's focus has since expanded to include studies like that of Salter and Murray (2014), who explored Adobe Flash and its impact on web design. The field of platform studies demonstrates the potential for rhetorical code studies in its goal of critical investigation into the relationship between code, design, user experience, and technological infrastructure.

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, such as 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 imposed 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, 27–45)

While not all software critics are interested in the technical qualities of digital media included in Manovich's list, these concerns nevertheless have informed much relevant scholarship, such as Cramer's (2005) analysis of algorithm as "magic," in which he broke down the ways various cultures have attempted to comprehend computation.

Several scholars have attended to the relationships between software, code, and infrastructure. Hayles' (2004, 2005) comparison of natural and code languages for meaningful purposes suggested that code, despite its mutable, transcodable nature—one that 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 and 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. Helmond (2013) examined the "algorithmization" of hyperlink construction and dissemination over time as expectations changed for web and social media navigation, sharing, and tracking activities. Noble (2018) has called attention to the ways that search engines like Google reinforce sexist and racist cultural values (especially those harmful to black women) through their algorithmic mediation and presentation of search results to users. Johnson and Neal (2017) highlighted the growth of black code studies, a related field whose scholars explored the radical innovations by people of color to digital technologies and resistances to industrially and politically normative uses thereof.

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 subsystems of language; and Kitchin and Dodge's (2011) exploration of the multiple levels of activity of daily life into which software are incorporated, as objects, processes, infrastructures, and assemblages (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 toward the conventional domain of rhetoric. Some critics even put into practice the creative development of software that might more clearly illustrate its relationship with culture, such as the "QueerOS" project by Barnett et al. (2016), a speculative operating system for discovering and exploring "new pleasures and possibilities both online and off" (n.p.). This was proposed in response to a perceived "lack of queer, trans, and racial analysis in the digital humanities, as well as the challenges of imbricating queer/trans/racialized lives and building digital/technical architectures that do not replicate existing systems of oppression" (n.p.).

Other scholars of software focus primarily on the types of processes the logic of which fuels the use of software programs, such as the calculations that provide Google search results or the behaviors of computer-controlled video game characters, what Wardrip-Fruin (2009) has referred to as "expressive processing." Because of its focus on how relevant technologies enable and constrain software and user behavior, this field has much in common with the related field of platform studies. Scholars involved in this subfield 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, 156). While many software scholars explore the processes and narrative experiences of video games and works of digital fiction, Bogost and Montfort (2009) explicitly specified that the field is not constrained to games as objects of inquiry, noting that even programming languages possess 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. While technical knowledge of computer systems has rarely been argued as *necessary* to perform successful inquiries into software processes, most scholars involved in expressive processing suggest that such a skill set is crucial to pursuing more fully questions of the subfield. 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" (158). That is, 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. Schmidt (2016) has offered a similar call to action for digital humanities scholars more broadly: "the first job of digital humanists should be to understand the goals and agendas of the transformations and systems that algorithms serve so that we can be creative users of new ideas, rather than users of tools the purposes of which we decline to know" (n.p.). Scholars studying software contribute to rhetorical code studies through their emphasis on the technological components integral to the humanistic analysis of digital media as well as how those components facilitate meaningful action for further invention and exploration.

Technical Communication

While technical communication was not identified as a foundation for rhetorical code studies, it nonetheless has had a long and rich relationship with rhetoric and software development that is worthy of mention, thanks to its focus on (among others) clear and effective communication that facilitates complex activities. While the majority of relevant technical communication scholarship has focused less on the explicit composition of code than on the composition and design of supporting communication practices and documents (e.g., installation guides, new user tutorials, etc.), it nonetheless contributes to rhetorical code studies through its emphasis on expert authors' need to answer less informed audiences' questions and address potential confusions with a given text or related activity. Further, given technical communication's close relationship to technology-oriented industries, scholarly projects in technical communication often offer unique insight into the development and dissemination of relevant texts, thanks to technical communication scholars' direct collaboration with industry professionals.

In particular, the work of Spinuzzi has been perhaps the most directly related scholarship in technical communication to rhetorical code studies. Spinuzzi (2002a) has explored a number of questions centering on how activity theory and rhetorical genre studies can illuminate, inform, and mediate professional programming practices. Spinuzzi (2003) examined software development as a kind of *activity system* and *genre ecology*, two means of understanding complex, interrelated acts of communication made up of numerous genres, the individual actors and communities who engage them, and the domains of knowledge necessary to do so. Most significantly, Spinuzzi (2002b) examined software code languages through the lens of paralogic rhetoric, viewing code as "a collaborative tool meant to help programmers share and review their work with others" rather than as a purely instrumental, machine-oriented set of commands (n.p.).

Similarly, scholars like Hart-Davidson et al. (2008) and Johnson (2014) have investigated the relationship between rhetoric and protocol (e.g., workflows), represented in systems like those of information or institutional infrastructure, in order to understand how technical communicators—among others—might effect change in regards to such systems. Warnock and Kahn (2007) considered the ways that informal and self-directed exploratory writing practices might impact programming practices as a means of more clearly tying together programmers' approaches to writing and thinking. Maher (2011) highlighted the relationship between software documentation-related literacies and the "evangelism" through which particular software ideologies (e.g., open source) develop and are expressed.

Others have continued to explore questions of genre and activity in relation to practices of software development and related knowledge work. For example, Applen (2001) examined knowledge management and XML authorship to communicate meaningful information in particular ways through metadata, as well as the data it describes, for particular audiences. Truscello (2005) called attention to the liminality of software through what he called the "*rhetorical ecology of the technical effect*[, which] marks the convergence of everyday life, the materiality of technology, and the web of cultural practices that constitute software" (349). Dyehouse

(2007) investigated the role of knowledge content analysis in regards to technological development, specifically how arguments were composed and delivered to nonexpert or nonspecialist audiences as well as how technical communicators could more effectively study such composition and delivery. More recently, Divine, Ferro, and Zachry (2011) studied a set of Web 2.0 services in order to learn about how communicative genres were developed and employed for a variety of knowledge work contexts. Swarts (2011) examined how technological literacy functions as a process through which social networks are constructed, developing a heuristic in order for "the kinds of rhetorical articulations that technical communicators create" to be better understood (297). Further, Swarts (2015) considered the procedures and processes involved in seeking and evaluating help online in regards to navigating issues of uncertainty and contingency that may affect how a problem is or could be solved.

The close relationship between technical communication and the software industry, as well as that of technical communication and rhetoric, provides rhetorical code studies with a rich body of scholarship and practice on which to draw, for inquiry not only into professional practices but also for comparative study with amateur practices. Given that technical communication scholars are increasingly focusing on software development in complement to end-user experiences or genres, it is likely that the field of technical communication will provide some of the most valuable conversations and investigations for those interested in exploring more fully the questions central to the rhetorical study of code.

Rhetorical Code Studies' Gains and Contributions

The field of rhetorical code studies exists within the territory I have begun to locate over the course of this chapter. 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 toward technological execution (the computation itself) and what sorts of expression that execution results in. But arguably the 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 construction of meaning 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.

Rhetoric, and digital rhetoric in particular, offers rhetorical code studies an established grounding in the study of meaning making and, with it, suasive action. 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 and through digital media. Special attention should be paid to rhetoric's tradition of focusing on the means by which rhetorical agents attempt to induce specific audiences to various kinds of action. 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 turn, rhetorical code studies provides rhetoric with a more focused and robust understanding of how code, software, and technological infrastructures serve to make meaning, directly and indirectly, in a wide variety of contexts.

Critical code studies is valuable to rhetorical code studies through its 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 might 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. Rhetorical code studies offers critical code studies a demonstration of its methodological strengths and flexibility, incorporating into critical code studies' fold the vocabulary and theoretical frame of rhetorical criticism to complement its existing literary foundation.

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 toward 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. Software scholars bring to rhetorical code studies an emphasis upon the malleable, computable, and inherently *meaningful* nature of digital data. Platform studies scholars call attention to the specific circumstances of a given computer technology and the software it runs, situating both within a particular cultural and historical context that can help us understand the decisions made to develop both, along with the implications those decisions may have had on the construction of subsequent technologies. Software critics have helped establish a space for rhetorical code studies by drawing attention to the performative and meaningful qualities of software designed for specific ends. As with critical code studies, a rhetorical approach to code provides software studies with another means of and language for articulating many of the relationships between code, software, procedural expression, platform, and user, along with a rich body of scholarship that has examined similar complex relationships and communication systems.

Rhetorical code studies has emerged from points of convergence among these fields, and it owes much to each for its theory and critical practice. At the same time, scholars examining code rhetorically have begun and continue to demonstrate the incredible potential that this area of study can offer back to those same fields and to others with overlapping objects of study. Building on the foundation that rhetorical code studies has established, the next chapter examines the longstanding relationship between algorithmic procedure and humanistic expression in order to illuminate even more fully how algorithms have been wielded rhetorically through history and how that historical use informs contemporary software development practices.