Rhetorical Code Studies: Discovering Arguments in and around Code

Book · March 2019

DOI: 10.3998/mpub.10019291

CITATIONS
6

READS
471

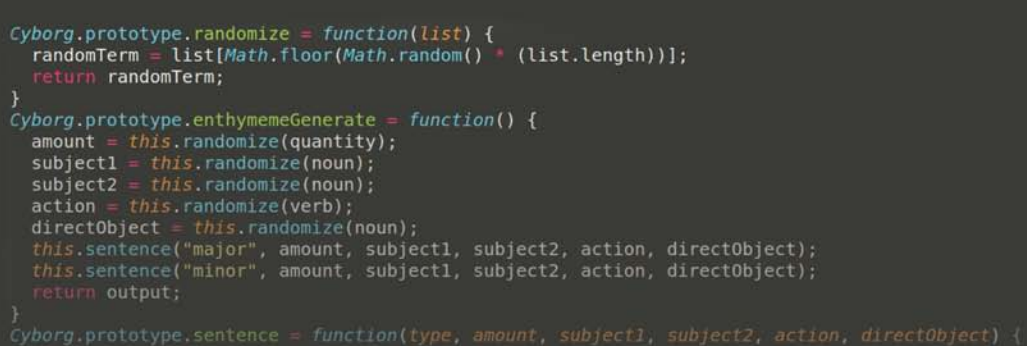
1 author:

Kevin Brock
University of South Carolina
10 PUBLICATIONS **50** CITATIONS

SEE PROFILE

rhetorical { code } studies

discovering.arguments in.and.around.code



```
Cyborg.prototype.randomize = function(list) {
  randomTerm = list[Math.floor(Math.random() * (list.length))];
  return randomTerm;
}
Cyborg.prototype.enthymemeGenerate = function() {
  amount = this.randomize(quantity);
  subject1 = this.randomize(noun);
  subject2 = this.randomize(noun);
  action = this.randomize(verb);
  directObject = this.randomize(noun);
  this.sentence("major", amount, subject1, subject2, action, directObject);
  this.sentence("minor", amount, subject1, subject2, action, directObject);
  return output;
}
Cyborg.prototype.sentence = function(type, amount, subject1, subject2, action, directObject) {
  if (type == "major") {
    myQuantity = amount;
    mySubject = this.singularize(subject1);
    myVerb = action;
    directObject = this.singularize(directObject);
    majorPremise = myQuantity + " " + mySubject + " " + myVerb + " " + directObject + ".";
    majorPremise = majorPremise[0].toUpperCase() + majorPremise.substring(1);
    output = majorPremise;
  } else if (type == "minor") {
    mySubject = subject2;
    myVerb = this.singularize(action);
    directObject = this.singularize(directObject);
    minorPremise = mySubject + " " + myVerb + " " + directObject + ".";
    minorPremise = minorPremise[0].toUpperCase() + minorPremise.substring(1);
    output += " " + minorPremise;
  }
}
```

+ kevin brock

RHETORICAL CODE STUDIES

SWEETLAND DIGITAL RHETORIC COLLABORATIVE

Series Editors:
Anne Ruggles Gere, University of Michigan
Naomi Silver, University of Michigan

The Sweetland Digital Rhetoric Collaborative Book Series publishes texts that investigate the multiliteracies of digitally mediated spaces both within academia as well as other contexts.

Rhetorical Code Studies: Discovering Arguments in and around Code
Kevin Brock

Developing Writers in Higher Education: A Longitudinal Study
Anne Ruggles Gere, Editor

Sites of Translation: What Multilinguals Can Teach Us about Digital Writing and Rhetoric
Laura Gonzales

Rhizcomics: Rhetoric, Technology, and New Media Composition
Jason Helms

Making Space: Writing, Instruction, Infrastrucure, and Multiliteracies
James P. Purdy and Dànielle Nicole DeVoss, Editors

Digital Samaritans: Rhetorical Delivery and Engagement in the Digital Humanities
Jim Ridolfo

DIGITALCULTUREBOOKS, an imprint of the University of Michigan Press, is dedicated to publishing work in new media studies and the emerging field of digital humanities.

Rhetorical Code Studies

DISCOVERING ARGUMENTS IN AND AROUND CODE

Kevin Brock

University of Michigan Press
ANN ARBOR

Published in the United States of America by the
University of Michigan Press
Manufactured in the United States of America
Printed on acid-free paper
First published February 2019

A CIP catalog record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data

Names: Brock, Kevin, author.
Title: Rhetorical code studies : discovering arguments in and around code / Kevin Brock.
Description: Ann Arbor : University of Michigan Press, [2019] | Series: Sweetland digital rhetoric collaborative series | Includes bibliographical references and index. |
Identifiers: LCCN 2018043066 (print) | LCCN 2018049477 (ebook) | ISBN 9780472131273 (hardcover : acid-free paper) | ISBN 9780472125005 (ebook) | ISBN 9780472901043 (OA)
Subjects: LCSH: Coding theory. | Rhetoric—Data processing. | Software engineering—Psychological aspects. | Computer algorithms—Psychological aspects. | Online social networks.
Classification: LCC QA268 (ebook) | LCC QA268 .B76 2019 (print) | DDC 005.13—dc23
LC record available at https://lccn.loc.gov/2018043066

https://doi.org/10.3998/mpub.10019291

The publisher gratefully acknowledges the support of the Sweetland Center for Writing in making this book possible.

To Erin, my best friend and partner in all things.

Acknowledgments

This project, like so many of the cases discussed in the following pages, could not have been possible without myriad collaborators, colleagues, and mentors, all of whom have profoundly impacted my life through their assistance and friendship. I cannot overstate the gratitude I feel for all the time, energy, and consideration that has been shared with me. At the same time, I recognize my words here will be inadequate in acknowledging everyone who has helped me and in describing the impact they have had on me and this book.

I want to begin by offering my deepest thanks to Nancy Penrose, who took a chance on me when I was a struggling master's student who felt ambivalent toward writing and its study. At the time, I did not know just how significantly my life would change by turning my focus toward rhetoric and composition. Without Nancy's support and patience with my gaining a grasp of the field, I would never have been able to pursue this career trajectory.

I also owe a considerable debt to the faculty with whom I studied in the Communication, Rhetoric, and Digital Media doctoral program at North Carolina State University; their guidance helped me develop the ideas central to my argument throughout this book. David M. Rieder offered invaluable and enthusiastic input throughout my time in the program. Alongside David, Susan Miller-Cochran, Jason Swarts, and Kenneth Zagacki all shared with me their incredible insight and feedback on my dissertation, which served as an early attempt at articulating my argument in this book. In addition, the courses I took with Carolyn R. Miller, Chris Anson, Susan Katz, Kelly Albada, David Berube, and Denise Gonzales Crisp helped me further refine the theories and methodologies informing my approach to communication, code, and software development.

While at North Carolina State, I was lucky to explore interdisciplinary inquiry alongside an excellent body of emerging scholars in the CRDM program, especially Jen Ware, Wendi Sierra, Dana Gierdowski, Robin

Snead, Christopher Cummings, Lauren Clark, Seth Mulliken, Kati Fargo Ahern, Jordan Frith, Jason Kalin, David Gruber, Matt Morain, Dawn Shepherd, Kathy Oswald, Shayne Pepper, Ruffin Bailey, Kelly Norris Martin, Anna Turnage, Heidi von Ludewig, Ashley Rose Mehlenbacher, Kate Maddalena, Fernanda Duarte, Brent Simoneaux, Molly Hartzog, Jeff Swift, Elizabeth Johnson-Young, Keon Pettiway, Alex Monea, Elizabeth Pitts, Josh Reeves, and Gwendolynne Reid.

My colleagues at the University of South Carolina have been a source of tremendous intellectual and moral support. Byron Hawk, Hannah Rule, John Muckelbauer, Chris Holcomb, Christy Friend, Pat Gehrke, Gina Ercolini, Misty Fenske, and Erik Doxtader have guided me through this project and the balancing act that an early-stage academic career entails. Undergraduate and graduate students at South Carolina, in and out of the classroom, have also contributed to my crafting a fuller articulation of my arguments in this book. They include Gerald S. Jackson, Cynthia Bateman Jackson, Trevor C. Meyer, Adam S. Lerner, Ragan Glover-Rijkse, Ben Harley, Jonathan Maricle, Adam Padgett, Kelly L. Wheeler, Sebastian Ivy, Amber Lee, Ashley Walker, and Christian Smith. I am grateful to you all for allowing me to discuss my research with you during our time together.

Over the past decade, numerous colleagues from across the United States have graciously contributed to the continued development of my research. Jim Brown and Annette Vee have been mentors whose input greatly impacted more of my work than I could concisely describe, while Doug Eyman, Chris Lindgren, Estee Beck, Steve Holmes, Kristin Arola, Casey Boyle, Liz Losh, John Gallagher, Mark Marino, Cheryl Ball, Karl Stolley, Bill Hart-Davidson, Ryan Omizo, Brandee Easter, Collin Brooke, and Rachael Sullivan have all challenged me in productive ways to refine my arguments and explore new directions for the rhetorical study of software. I am grateful for their many thoughtful engagements with me. I also especially want to thank Matt Davis for the innumerable hours he spent reading and reviewing my drafts. This would be a much weaker book without his keen and considerate insight.

Naomi Silver, Anne Ruggles Gere, the members of the Sweetland Digital Rhetoric Collective editorial board, and Mary Francis and the University of Michigan Press provided me with insightful and compassionate support for this project and its publication. I am also greatly indebted to the reviewers who examined an initial draft of the manuscript, as their feedback guided my argument to evolve considerably and in ways I had

not initially anticipated. I did not ever think I would enjoy so much working with a press.

Finally, I want to thank my patient friends and family for supporting my efforts and tolerating my distant- and absent-minded behavior while I worked, especially my wife Erin and my mother Cindy. David, Patrick, and Jim Brock were reliable sources of needed levity and distracting, although always informative, debate. My friends Jonathan, Adrianne, Gabriel, Scott, George, Pat, Dustin, and Ibrahim always made themselves available for whatever assistance they could provide. I don't know how you all have put up with me, but I will always adore you for it.

Contents

Digital materials related to this title can be found on the Fulcrum platform via the following citable URL: https://doi.org/10.3998/mpub.10019291

Tables

Practice Scripts

Figures

Introduction

For several days in May 2017, approximately 200,000 computer systems were infected by "WannaCry," a ransomware attack that exploited a set of security vulnerabilities in the Microsoft Windows operating system. Once WannaCry gained access to a new system, it would check if a given website domain was registered. If the domain was not registered, WannaCry would encrypt data on the system's drive(s) and then propagate itself randomly to other systems via the Internet and any local network connections. Then, WannaCry displayed a message to users of the system that their data was being held hostage and that a ransom could be paid via Bitcoin payments to specific recipient addresses (Khomami and Solon 2017; Lee et al. 2017).

While security patches were quickly developed and distributed to fix the vulnerabilities WannaCry exploited, a pair of key clues regarding its functionality and authorship were identified by attending to WannaCry's algorithmic activity. First, an anonymous individual, recognizing that the ransomware attempted to contact a nonexistent domain as part of its initial activity, registered that particular domain name. The effect was that of a "kill switch," immediately stopping thousands of WannaCry iterations *per second* from continuing on to their encryption processes (Khomami and Solon 2017). Second, the ransom messages displayed by WannaCry were analyzed by linguistic experts, who determined that the program's author(s) were most likely nonnative English speakers who used automated translation software (Leyden 2017). Further, the analysts identified the authors as most likely being Chinese in origin, although North Korea was later credited with its construction (Bossert 2017).

Both of these interpretive responses to WannaCry highlight the rhetorical nature of the ransomware program and, by association, of software and code more broadly. A consequence of registering the "kill switch" domain transformed WannaCry's ability to propagate across networks and

hold further users' data hostage. An identification of the authors' location facilitated collaboration among international intelligence organizations as well as cybersecurity experts, in case political action became necessary to resolve the situation. The rhetorical dimensions of all these acts—of WannaCry's composition and proliferation as well as the analyses and responses that emerged to understand and combat it—can tell us much about how to assume more nuanced and ethical approaches to creating, using, and discussing software in public, profession, and academic contexts alike.

Understanding code as rhetorical and not merely instrumental is not new in the fields of rhetoric and technical communication, however. Nearly forty years ago, Miller (1979) argued that technical writing should be understood and taught not as a *merely* instrumental skill set for communicating information (that is, as a kind of transparent and neutral vehicle for information) but instead as a highly rhetorical means of creating knowledge that allowed writers to construct and participate in particular communities. Miller suggested that the ways technical documents were composed and designed were as important and inherently meaningful as the content described therein.

We can recognize a similar exigence in current scholarly and public conversations regarding digital media and their uses, from WannaCry to far less terrifying contexts. Such conversations tend to focus on the capacity of software for incredible social and political change, from the role of social media in organizing protests during the "Arab Spring" of 2010–2012, to the impact of high-frequency trading on the global market, to the uncannily precise targeting by various companies (e.g., Target or Amazon) of individual consumers with customized advertisements and sales. Other capacities for change include broadening access to technologies that augment individuals' abilities to communicate in diverse ways with larger and more geographically distributed audiences, including global positioning system technology, YouTube videos, peer-to-peer file sharing networks, 3-D printers, and online financial transactions. Alongside the conversations about these media, others also occur that focus on education and vocational training, with an almost singular goal of building a larger and, in many cases, explicitly more diverse and inclusive population of future programmers whose skills will certainly be valuable in a software-driven world.

Unfortunately, outside of specific circles, these various conversations on digital media—whether education-oriented or not—tend to focus on

software as an instrumental tool and thus ignore or otherwise fail to address the role that meaning making, and in particular meaning made *in and around code*, plays in the development and use of software. Code, however, can also be approached rhetorically and critically in reflection of its meaningful nature. An acknowledgment of the roles that code and its authors play is important for moving forward in any of these aforementioned conversations, not only for identifying what has happened or is happening but also how to induce desired change to the status quo.

It is often only after the fact that various social, cultural, and political problems reflected in how a given software program has been designed to function—whether recognized beforehand or not—are publicly acknowledged and addressed. Among the myriad examples of digital fiascos and *post hoc* revelations about their subjects' limitations include the Nikon camera software that told Asian users they were blinking in photos (Rose 2010), with the software allegedly not taking into account different ethnicities' facial structures. In another example, Facebook's "graph search" technology enabled users to conduct potentially disturbing surveillance of their neighbors, such as combining searches for "married people" with those users who liked "prostitutes" and providing searchers with the ability to contact those users' spouses (Brock and Shepherd 2016). "Tay," a Twitter bot developed by Microsoft, shortly after its activation began to post racist content to other users of the platform. It was discovered that the bot had built a vocabulary of hate speech from public Twitter data, with a substantial amount of hateful language directed toward Tay specifically to "teach" it that language (Mason 2016). In each of these cases, it is impossible to be certain about the extent to which these issues may have been anticipated, but we have an ample supply of public backlash and criticism over the released versions of these programs and the impact they had on their audiences.

Discussions among scholars of rhetoric, technical communication, and software studies about digital media and technology tend to be exceptions to the popular conversations described above in that scholarly discussions stress critical reflection on the social and cultural implications of particular media or situations wherein they are used. But such discussions infrequently focus *directly* on the software used for particular media and even less frequently on the code that drives that software. Further, there is a divide separating those voices who would increase access and exposure to the instrumental application of programming-related education and those voices who would facilitate critical and rhetorical awareness and

employment of code and its procedural logic for particular civic, political, and economic ends, attending to the myriad dimensions that influence and are influenced by the construction of software programs.

This is not to say that efforts to bring together these groups of conversations about critical, rhetorical, and instrumental approaches to studying digital technology are not attempted more broadly. Indeed, a great many public opinion pieces by media critics, programmers, and educators have championed various fusions of instrumental code composing or "making" with critical reflection on those composing activities. Similarly, numerous programming platforms have been developed to make software development more accessible and palatable to wider user populations than might otherwise encounter them, such as Scratch, Codecademy, Processing, or Hackety Hack. As Ford (2015) explained, it tends to be much easier to develop more accessible ways to teach people existing languages than to develop new languages or change existing ones:

> Making a new [programming] language is hard. Making a popular language is much harder still and requires the smile of fortune. And changing the way a popular language works appears to be one of the most difficult things humans can do, requiring years of coordination to make the standards align. Languages are large, complex, dynamic expressions of human culture.

Ford's description of programming languages as "large, complex, dynamic expressions of human culture" is incredibly important. Ford emphasized the influence of *human culture* on the creation of such languages, since their incorporation—and exclusion—of particular values, perspectives, and styles are often elided in discussions of particular languages' computational efficiency or "elegance," terms examined in more detail later but that, generally speaking, emphasize technological speed over human dimensions of influence on the composition of a given code text. Of particular note for efforts like Scratch and others is the broad push to reframe STEM (science, technology, engineering, math) education as STEAM (science, technology, engineering, arts, math) so as to build on the widespread support for STEM initiatives. These initiatives, however, tend to avoid direct engagements with rhetoric and code in favor of its political or cultural impact in a given context.

This latter avoidance of direct engagement with code (as both text and activity) is of central significance to the argument I make throughout this

text, as I want to draw attention to the rhetorical activities in which programmers engage, in and around code texts, as they develop software.[1] Among the most important public outreach tasks a rhetorician who studies digital technologies can engage in is helping diverse audiences understand not only how those technologies influence audiences toward particular actions but how those same technologies *are designed and constructed* to influence them. Among the most powerful ways to do this, I suggest, is to draw attention to the appeals and strategies employed in the construction and dissemination of software code among programmers of varying expertise and involvement in a given project. It is these individuals and groups, after all, who decide not only what a program will do but how they will go about making it perform those tasks. The communication that takes place in the lines of code they write—along with communication in code comments, emails, bulletin board posts, and other venues—illuminates a great deal about the kinds of meaning programmers can and choose to create in and through code. Given the ubiquity of digital technology for daily tasks and phenomena, it is imperative that rhetoricians work not only to recognize what occurs rhetorically in the current paradigm of programming education and exercise but to become involved in its development toward a more explicitly and intently meaningful form of communication.

In this book, I demonstrate how a shift in code-related orientation toward the rhetorical opens up new opportunities for critical inquiry, as well as how this shift is already taking place. In chapter 1, I map the various academic conversations in related fields within the humanities that have some overlap with the study of rhetoric and digital media, including software studies, critical code studies, and technical communication. From this mapping, I argue for the further cultivation and exploration of an emerging field, which I call "rhetorical code studies," that centers its focus on considerations of software code as a form of and site for rhetorical communication.

In chapter 2, I examine the historical relationship between algorithms and rhetorical invention that extends back far beyond digital technologies.

1. Throughout the book but primarily in chapters 3 and 4, I discuss posts and code excerpts by a number of users on various software versioning system-related websites. I refer to the real names of those users who are either very well-known in the industry or whose usernames are identical to their real names; otherwise, I refer to users by the username they employ on the website. While in some instances a real name might be inferred, I have opted to call users by the monikers by which they wish to identify themselves on those sites.

That is, we have a long tradition of critical and humanistic approaches to mathematics and logic on which to build a rhetorical understanding of code and the computation it describes. From this foundation, I examine how the procedural logic of algorithms functions for rhetorical ends in software code, and three example types of programs demonstrate some, but hardly all, of these ends in action.

Chapter 3 is centered on the discourse that surrounds code, such as code comments (lines of text in code that are not interpreted by a computer), email conversations, and other forums dedicated to development project discussions. If code functions as meaningful communication, as I posit it does—inherently—then the sorts of meaning making that occur in genres connected to and surrounding code can offer insight into programmers' goals for particular computational tasks and types of actions they intend to induce through the use of their software. While there is not always a 1:1 relationship between what is said and done in code with what is said and done about it, we can nonetheless learn much by examining what connections can be recognized.

In chapter 4, I build on the argument established in the previous chapter to explore a case study of *some* components of the code for Mozilla Firefox, a large-scale software project developed regularly over the past two decades by hundreds of programmers. Given the sheer amount of code written during that period, I focus on a handful of rhetorical tactics and goals present in the browser's code to see how its authors attempt to communicate particular kinds of meaning to their collaborators about how the code operates (or how it is meant to operate).

In chapter 5, I turn from analysis of existing practices to composition of a set of example programs or "practice scripts" to emphasize their use as starting points for further experimentation with code. Specifically, I connect the historical rhetorical exercises of *progymnasmata* with small-scale programming tasks as one potential and initial means of realizing an approach to software development that is explicitly informed by rhetoric. These exercises may help readers unfamiliar with programming begin to recognize and experiment with employing particular rhetorical concepts for specific procedural ends.

Finally, in chapter 6, I consider how rhetorical code studies might be further developed in several significant scholarly directions, including—for educational initiatives—how rhetorically informed efforts to program might be assessed by instructors, just as other forms of assessment have been developed for composition in multiple and diverse modes.

Ultimately, I hope that the close relationship I identify in this text between rhetoric and code will be taken up and explored further by other scholars, whether in rhetoric or in other fields. Perhaps a rhetorical study, and practice, of code can lead to more software development engaged with the ideologies and values of programmers and users and that works to effect cultural and political change more fully, actively, and explicitly to the benefit of those same populations. For a world fundamentally impacted by digital technology—and thus by its software—the arguments we and others make in code have the potential to be read, experienced, and responded to so that we can develop not only more responsible and ethical *programs* (to use a term coined by Brown 2015) but more responsible and ethical *publics* as well.

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 en-

sures 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 gender-

ing 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 rhetori-

cal 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 socio-

political 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 underly-

ing 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 schol-

ars 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.

CHAPTER 2

Rhetoric and the Algorithm

The algorithm is perhaps the concept most central to rhetorical code studies, and it is necessary to examine how algorithmic procedures are related to humanistic scholarship in general and to rhetoric in particular. This relationship can be demonstrated by tracing 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. Following this brief history of the algorithm and its connection to humanistic work is an interrogation of how the algorithm plays an integral role in rhetorical activity. Such activity can be understood from a perspective that Hayles (2012) has referred to as "technogenetic," meaning that it identifies the interrelated codevelopment (or, for Hayles, coevolution) of human and technological entities (10). For rhetoricians, this means not *only* digital or electronic technologies but all apparatuses, broadly employed, for the purpose of making and communicating meaning as well as the "specific implications" Gillespie (2014) has identified "when we use algorithms to select what is most relevant from a corpus of data composed of traces of our activities, preferences, and expressions" (168). Algorithmic construction of meaning, the execution of a kind of "knowledge logic" (Gillespie 2014, 168), works to facilitate action in a variety of digital contexts, emerging from their predecessors in new and familiar ways.

From Algorithm to Algorithmic Culture

An algorithm is, in broad terms, a procedural framework for accomplishing a particular task. Understood simply, it is the description of a task-oriented procedure through its component operations (i.e., its steps). The algorithm's most common explicit disciplinary usage occurs in engineering, computer science, and mathematics, where the algorithm exists as

a procedure with a discrete number of tasks whose operations make use of clearly defined conditions that impact subsequent decisions within the procedure. For example, the algorithm for a web page to display a certain time of day is likely to rely on determining where, geographically, the IP address for a given user's computer exists: if it is on the Atlantic coast of the United States, the time's display may accordingly adjust to UTC-5, or five hours behind Greenwich Mean Time. An algorithm for washing one's clothes may involve a condition wherein the washer is cleaning a load of white or colored garments; if the former, the algorithm may involve a step regarding the inclusion of bleach into the mix whereas washing colors would not involve that step.

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). As noted by Steiner (2012), however, the algorithm *as a concept* predates al-Khwarizmi's work, or formal mathematics in general, by several millennia. As Steiner observed, the algorithm—procedural activity—existed long before al-Khwarizmi explicitly described the algorithm as a concept involving procedure. Even though it was not necessarily defined as a clear concept until al-Khwarizmi established it, Steiner argued, the algorithm has played a number of important roles in daily or common cultural activities for millennia:

> 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 [. . .] have used them in an attempt to predict the future. (Steiner 2012, 54)

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

For al-Khwarizmi and for mathematicians since, the algorithm was the procedural framework through which a mathematical equation would be

calculated. By constructing a framework to which a mathematician could adhere in order to solve discrete problems, the capabilities of symbolic systems to reflect logical procedures were suddenly clearly articulated. One example of al-Khwarizmi's algebraic algorithm in action demonstrated how a mathematician could 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, 7). In mathematical notation, this equation can be demonstrated in the following steps:

Step 1: $5x^2 = 80$
Step 2: $x^2 = 80 * 1/5$
Step 2a: $x^2 = 80/5$
Step 3: $x^2 = 16$

The algorithm can be extended further to determine the value of the square root:

Step 4: $\sqrt{x^2} = \sqrt{16}$
Step 5: $x = 4$

The procedure to determine the value of x^2 involves condensing as many relevant operations of the equation as possible so as to calculate quickly and accurately the numerical value of x^2. While the symbolic mathematical notation by which algorithms could be most efficiently expressed was not developed until several centuries after al-Khwarizmi, the potential of algorithmic power for analytical and utilitarian employment had been clearly 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. In the mid-nineteenth century, Ada Lovelace would lay out a vision for the potential of computers to operate by means of programmed (algorithmic) instructions:

> A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible. Thus not only the mental and the material, but the theoreti-

> cal and the practical in the mathematical world, are brought into more intimate and effective connection with each other. (2002, 19)

Lovelace's description of a possible language through which to manipulate computer technologies (and specifically Charles Babbage's "Analytical Engine") refers implicitly to algorithmic procedure for the sake of mathematical analysis. What had not yet developed at the time, but which would emerge just after her death, was a clear logic to drive algorithmic grammars toward practical ends.

Modern-day 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 other arbitrary comparison states, e.g., "on" and "off," or "1" and "0"). Shannon (1937) demonstrated how electrical circuits, by being opened or closed, could serve as a viable 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 technologies.

By checking the state of one or more given data elements within a computational system—what is referred to as "input"—a Boolean algorithm can allow a software program to execute particular computational tasks so as to express a relevant "output" body of data. 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)" (4). In other words, computer science makes significant use of Boolean-powered algorithms in part because algorithms, especially procedures that can be *automated* by a computational system, align effectively with the Boolean foundation upon which computers and electronic 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 a wide range of 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 identified the algorithm as a paradigm-shifting phenomenon whose importance has had

world-changing effects. For Steiner, the increasing control that complex computer algorithms possess in contemporary culture is critically significant, as those who can create, understand, and manipulate complex algorithms with digital technologies have arguably become a new ruling class (an argument that has been taken up by other critics, such as Rushkoff 2011). Across these texts, there is a shared central premise that algorithms, especially those meant to be expressed via computer technology, are quickly gaining—or already possess—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.). It may be accurate to say that to be aware of algorithmic procedure or to be able to work *with* algorithms is to be able to influence the trajectory of social, cultural, and political development to extents far beyond those phenomena that can be influenced by individuals who are unaware or uninvolved with algorithmic procedures.

While the vast majority of academic and professional discourse related to algorithms—which has overwhelmingly taken place in the disciplinary spheres of mathematics, computer science, and engineering—has focused on computational algorithms, the conceptual constraints surrounding algorithmic procedures tend to vary from discipline to discipline; the specific field in which a scholar or practitioner works has some influence upon how the scholar is likely to approach algorithms and their potential for certain tasks. This understanding is important as the particular language used by a scholar or practitioner to describe and explain what an algorithm *is* and *does* illuminates the recognized potential(s) that its author, or his or her disciplinary community, attributes to algorithmic procedure.

For example, computer science tends to base 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 and Bratley (1996) defined an algorithm as "a set of rules for carrying out some calculation, either by hand or, more usually, on a machine" (1). For Hillis (1998), it is "a fail-safe procedure guaranteed to achieve a specific goal" (78). Black (2007) defined the term as "a computable set of steps to achieve a desired result" (n.p.). Clearly, there is a conventional understanding of the algorithm as relating to replicable procedures made up of discrete operations to be executed through a computer or with its assistance. But not all scholars in the field describe algorithms in such 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. (1)

Edmonds is quick to separate algorithm as concept from the code-centric applications of algorithmic procedure for computer science (e.g., software), organizing the latter within the hierarchy of the former; as a result, through the expanded definition of the term, the possibilities of the algorithm beyond the scope of computer science become much more situational and flexible than might otherwise be conventionally assumed.

Other fields with less direct foundation in mathematics often more readily accept the algorithm to possess these more fluid qualities than do those fields relevant to computer science. Accordingly, the means by which algorithms are approached and used for ends in these other disciplines are quite different as well. Ramsay (2011) described the algorithm as a concept in relatively flexible terms as "a method for solving some problem" (18). This definition has some significant overlap with that of *heuristic*, generally defined as a broad framework for problem solving and which, in rhetoric, is tied closely to the canon of invention through its emphasis on discovery (Herrick 2016, 27). While such overlap is not inherently problematic, the less "clear" a problem-solving method becomes, the more difficult it may be to reach consensus on whether that method performs in either an algorithmic or heuristic sense.

Gillespie (2016), meanwhile, has identified a range of context-specific metaphors for algorithm and how its understanding is communicated: trick, synecdoche, talisman, and commitment to procedure. Each of these metaphorical associations, Gillespie suggested, indicate both an identification of a particular audience (one that is expected to grasp and accept said metaphor) as well as of the individuals and communities employing those metaphors, since they do so for specific purposes in order to induce their audience(s) to some relevant action. For Gillespie (2016), this employment is "discursive work [that] the term performs" while rhetors and audiences are forced to navigate its multiple meanings (18).

Each of these cases brings to mind Steiner's (2012) overview of the algorithm as procedure relevant beyond computer science to law, medicine, grammar, and other diverse efforts toward predicting the future. Through these definitions of algorithm as procedure intended *for purposefully solving*

problems, nonscientific disciplines bring to the conversation an emphasis on how humans approach the accomplishment of particular tasks (and not necessarily what those approaches will be in any given situation). In order to clarify what the significance of this conceptual shift means for the rhetorical study of algorithms, I will situate the applications of algorithmic procedure across fields within the humanities.

Algorithmic Criticism in the Humanities

An inclusive understanding of algorithms as problem-solving procedures certainly incorporates into its scope the computational algorithms that drive electronic technologies and computer software. But humanistic inquiry relating to algorithms focuses on how a particular algorithmic procedure reflects the goals and values of its developer(s) and on the means by which computational procedures facilitate novel approaches to critical engagement and meaning making rather than focusing on the technical expertise that conventionally accompanies particular forms and applications of computation. Such inquiry explores both the complex situations that algorithms impact *and* the situations in which certain algorithms are composed, including how those compositions are structured in order to make a particular engagement. For rhetoric, this combination involves multiple scales of rhetorical activity, from the exigences that spur the creation of a given body of code to the specific devices used to frame and describe any response(s) to those exigences.

Even though my focus here will ultimately consider how algorithms are useful and significant components of persuasion, especially in regards to contemporary rhetorical activities, I first want to 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 classical *enthymeme* to the more recent phenomenon of digitally mediated manipulations of massive data sets.

Humanistic Algorithms before and without Computers

The idea of procedure—whether explicitly connected to "algorithm" or not—as a means of generating or facilitating action has accompanied creative and critical practices since early uses of mnemonic devices to recite oral poetry; as Ong (2002) has noted, "In a primarily oral culture, to solve effectively the problem of retaining and retrieving carefully articulated

thought, you have to do your thinking in mnemonic patterns, shaped for ready oral recurrence" (34). As I explore below, the application of diverse algorithms for humanistic, and especially rhetorical, purposes remains a significant component of productive and critical activity, activity that reflects Nowviskie's (2014) description of algorithms employed for humanistic activity that "can be understood as problem solving and [. . .] as open, participatory, explorative devices" (151). The identification of an algorithmic humanities is integral to understanding how algorithms work for persuasive ends, a necessary requisite for examining algorithms and code as rhetorical communication.

ENTHYMEME AS ALGORITHM

As I have argued elsewhere (Brock 2014), the algorithm most central to Western rhetoric is the *enthymeme*, a concept that has predominantly been defined as an incomplete logical argument that, through its procedural logic and presentation, compels an audience to complete the argument in order for it to be properly and effectively expressed (Bitzer 1959; Walker 1994). Specifically, the enthymeme is a rhetorically oriented *syllogism*, a set of premises (major and minor) that, in combination, lead to some sort of conclusion or result. For a complete syllogism, the relationship between these elements is explicitly stated (see below). For an enthymeme, this relationship is *probable* and implicit, as its logic remains procedurally suspended until audiences identify it on their own. As Hairston (1986) has argued, "The person who argues from an enthymeme is [usually] not trying to *prove* a proposition; he or she is only trying to establish high probability" for an audience to accept the rhetor's proposition (76). 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, according to the statements' parameters (all *A* are *B*; *C* is *A*; therefore, *C* is *B*). More complex syllogisms can be constructed to create disjunctive or conditional reasoning. For example, the following is a disjunctive syllogism:

1. We will meet either in Paul's office or in the conference room. (Major premise)

2. We are not meeting in the conference room. (Minor premise)
3. Therefore, we are meeting in Paul's office. (Conclusion)

Conditional syllogisms generally include a determination of a condition being met. For example, one might say, "If it is nighttime, then one needs to drive with one's car's headlights on" 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 in varied ways to dramatically increase the flexibility with which one could frame the argument for 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 component logics might establish on its own. Carroll (1973) demonstrated a number of polysyllogisms as puzzles to be solved (what a rhetorician might reframe 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";
(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." (Carroll 1973, 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. Specifically, Shakespeare is a true poet [statement 5] (as he wrote *Hamlet* [statement 3]). As a true poet, he can stir the hearts of men [statement 2], which means he is a writer who understands human nature [statement 4]. Because he is such a writer, he is clever [statement 1]. The deductive process for Carroll's polysyllogistic example is less direct or linear than a simpler syllogism, but its computational nature is indisputable. One *cannot* move forward with any of these ideas until their relations have been appropriately sorted or processed. Indeed, many contemporary algorithms overwhelmingly rely on complex polysyllogistic reasoning to compute data dynamically in numerous iterations.

In contrast to a fully articulated syllogism, the enthymeme functions

algorithmically by leaving implicit reference(s) to one or more of the syllogism's components so that an audience will compute the logic of those missing components. Walker (1994) has hinted at the enthymeme as a kind of algorithm in his description of the enthymeme as the center of "argumentative or suasory procedure" (61). Similarly, Walton (2001) argued that the enthymeme suggests a "plausibilistic script-based reasoning" commonly explored in artificial intelligence research (93). One such example of the enthymematic algorithm is the following statement that recalls an example syllogism presented earlier in this chapter: *all humans are mortal; thus, Socrates is mortal*. This statement obliges 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, the enthymeme might be stated as follows: Socrates is a human and therefore is mortal. The reader's ability to follow this argument hinges on recognizing (i.e., processing) the implicit association that Socrates' 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. The audience "becomes" a computer in order to express this procedure by reaching the (likely anticipated) outcome. For developers, the enthymematic 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 constrained sense of "a method for solving some problem" (Ramsay 2011, 18) and instead demonstrates its fundamental flexibility and contingent nature as a process through which an audience is led to persuade itself to achieve action, via such means as deliberation and conditional deduction.

One trend that aids the emergence of rhetorical code studies is the relatively recent exploration by several rhetoric scholars of the bounds of enthymematic persuasion. Specifically, these scholars have examined whether the enthymeme can function as a rhetorical tool with value beyond the scope of conventional discourse. For instance, Smith (2007) 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 initially 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.

THE ALGORITHMIC RHETORICAL SITUATION-ECOLOGY

Another of the most significant algorithmic frameworks related to the study of rhetoric is that of the "rhetorical situation" (Bitzer 1968; Vatz 1968) and its more complex ecological rearticulations (Cooper 1986; Edbauer 2005). The rhetorical situation involves several interrelated components that together facilitate an effective rhetorical activity. First, a rhetor identifies a relevant *exigence* they wish to engage. An exigence is commonly defined as "an imperfection marked by urgency [. . .] something waiting to be done" (Bitzer 1968, 6). It is the catalyst for rhetorical, and any subsequent, action. Further, an exigence requires the capacity 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 is established, whether understood as "recognized" or "invented," that the algorithmic quality of the rhetorical situation can begin to be processed.

The second situational component is *audience*, the body of individuals that a rhetor hopes to induce to action, a body whose members "are capable of being influenced by discourse and of being mediators of change" (Bitzer 1968, 8). An identification of audience includes a recognition of the audience's values, background, and ideological leanings as well as of the modes of communication most likely to persuade the intended audience through their employment. 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 whether that approach it is the "objectively" most superior means of influencing an audience.

The third component is *constraint*, the limitations and influences exerted upon rhetor and audience alike as part of the expression of a rhe-

torical 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" (516). Rhetorical constraints include all restrictions upon a rhetor in how they might induce an audience to act, from modes of communication to particular suasive strategies or even specific language decisions.

Once an exigence has spurred a rhetor to act, and once that rhetor has identified an audience and the set of constraints framing the audience's potential reception of the provided argument, what comes next? The rhetorical situation's algorithm is expressed: the rhetor makes use of their chosen variables to bring about the intended outcome, that is, the change sought in regards to the spurring 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 contingent on the relevant exigence, audience, and constraints, and an audience may not always respond in the same way to a given rhetor or argument.

It is in this space for chaos, for unpredictable or unanticipatable responses, where the algorithmic character of the rhetorical situation becomes most intriguing. The rhetor anticipates how each situational variable *might* (or is *likely* to) influence the others, but this anticipation ultimately remains one influence of many upon the situation. It is only when the rhetor attempts to *express* their argument through the situational algorithm that any action, whether intended or not, can be achieved. Whether consciously or otherwise, a rhetor computes the procedure that emerges when *enough* of the situational variables have been identified to assemble an argument in pursuit of a particular goal.

Edbauer (2005) has observed that the rhetorical situation, as defined by both Bitzer (1968) and Vatz (1968), is problematic in its reduction of rhetorical activity to a single equation of exigence + audience + constraint + rhetor, since numerous situations interrelate at any time in a larger rhetorical *ecology* of diverse actors, motivations, and exigences. Edbauer's (2005) critique is significant for rhetorical code studies in that it draws greater attention to the complexity of the connected algorithmic procedures that make up acts of rhetorical communication. In other words, while it is important to recognize how rhetor, audience, constraint, and exigence influence one another in a bounded, individual situation, it is just as important to acknowledge that these dynamics are themselves components of an

even greater ecologically rhetorical algorithm affecting broader discursive trends across various agents and channels of persuasive potential.

Recognizing the flexibility of the rhetorical situation-ecology is integral to an understanding of code-based action as rhetorical. Specifically, it points to the indeterminacy 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 particular 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.

AESTHETIC AND POETIC ALGORITHMS: THE OULIPO

While I ultimately lead us toward a discussion of algorithms for rhetorical purposes specifically, I first want to discuss briefly some creative (aesthetic and poetic) approaches to algorithmic scholarship and production that 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 gathered under 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, 27). That is, the Oulipo acknowledged the possibilities of flexible and contingent meaning making within the framework of mathematical computation, specifically regarding how computation could be used to influence the composition of rhetorical texts. 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" (51). The composition of *structures*, complete with rules and constraints for successful invention within

those structures, is significant for this discussion because it emphasizes the possibilities of procedural expressions rather than the skill with which a particular expression is created in reflection of the nuances surrounding 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" (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) was considered superior by the Oulipo because its author(s) and readers alike could appreciate how its algorithmic procedures were expressed in order to produce the text "output." This is not to suggest that the limits of procedural constraint (the amount and range of expressions that could be produced) are necessarily *restrictive*. For example, 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 reader swapping in or out a given line from one of the sonnets with the appropriate line from 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, 118–21).

While the members of the Oulipo explored these structures to demonstrate the possibilities of invention through procedural constraint, it is also accurate to say that they highlighted the existing means of rhetorical invention and arrangement and offered new insight into these means 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" (18). In particular, Oulipian algorithms function rhetorically in that they emphasize the *potential* for constructing meaning through particular literary structures of constrained computation. This potential has even begun to be explored in depth with software code languages; Lopes (2014) built on Queneau's (2009) *Exercises in Style* in order to demonstrate stylistic influence on programming activities by following Queneau's example of writing the same vignette through the lenses of ninety-nine different literary styles. Lopes wrote the "same" program, a means of analyzing term frequency, in more than thirty different variations, based on the fundamental

stylistic concerns that guide each approach. As Lopes noted, "By honoring different constraints, we can write a variety of programs that are virtually identical in terms of *what* they do, but that are radically different in terms of *how* they do it" (2014, xii). Rhetoricians interested in the complexity of diverse possibilities that emerge from constrained situations—whether digital in nature or not—can take a great deal from the Oulipo's experiments with algorithmic approaches to invention.

Humanistic Algorithms in the Age of Computers

While algorithmic procedure and humanistic activity have been intertwined for much of human history, it is also true that computer technology has radically transformed this relationship. As a result, algorithm has, through code, developed beyond serving as a means for performing traditional rhetorical or critical action through the construction and execution of procedural activities. Specifically, it has become a form of meaning making in its own right. For computers, the algorithmic code that makes up digital software is rhetorically powerful thanks to its ability to engage data, machine, and human alike, albeit in different ways and for different ends. For most scholars, rhetorical engagement with and in code serves primarily as a tool to facilitate other iterative experiences, but when we recognize code as an algorithmic mode of communication, used specifically for rhetorical ends, we can approach a moment of clarity in which we reconsider how code influences us—humans and nonhuman technologies—to act through a quality that can best be described as algorithmic persuasion.

ALGORITHMIC CRITICISM

Possibly the most explicit 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 into novel "paratexts" that reveal insights that are otherwise unavailable or difficult to recognize. Paratexts, for Ramsay and other literary scholars interested in algorithmic criticism, are texts transformed, deformed, and performed in innovative ways and for various ends through procedural mutation and reconfiguration; 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, trun-

cated, and transduced" (2011, 16). Among the means by which Ramsay demonstrated the literary potential for algorithmic procedure is the aggregation and computation of particular types of data in order to restructure the literary reading experience.

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 in table 2.1 (Ramsay 2011, 13) is an excerpt from the expressed set of frequently used terms for two of the novel's six protagonists using a relevant algorithm employed by Ramsay. Ramsay's line of inquiry sought out the terms that were not only most frequently used *but also* were not used frequently by any other character, that is, the terms that were either unique to, or at least primarily associated with, each protagonist.

One conclusion related to new connections that Ramsay draws is that although the character Louis (as the only Australian in the group) is self-conscious of his accent, the other characters seem to pay that cultural distinction little attention: no one else, for instance, ever mentions the terms "Australian" or "accent." To the others, these concepts seemingly do not matter since they do not appear among the terms used frequently by those characters (Ramsay 2011, 12–14).

This sort of reading by Ramsay—in which previously unrecognized

Table 2.1. Excerpted lists of term frequency from Woolf's *The Waves*, compiled by Ramsay (2011)

Louis	Neville
mr	catullus
western	doomed
nile	immitigable
australian	papers
beast	bookcase
grained	bored
thou	camel
wilt	detect
pitchers	expose
steel	hubbub
attempt	incredible
average	lack
clerks	loads
disorder	mallet
accent	marvel

meaning is exposed through expressions of algorithmic arguments—demonstrates not only a new form of *reading* but also a new form of *research*: the development of a body of paratexts reflecting the logics of computational approaches to critical investigation and interpretation. In other words, the emphasis for algorithmic criticism should be placed as much on how relevant critical work occurs, and what happens in the process, as on what algorithmic criticism reveals through its individual, iterative expressions.

Ramsay's approach to criticism clearly identifies algorithmic procedure as a valuable tool to be used for scholarly criticism. Specifically, the use of algorithms to explore novel means of reading allowed Ramsay to generate paratexts that transform a given text or set of texts into new objects of study. When employed in this manner, the algorithm is a method for interpretation that, in part, quantifies those components of an original text that a critic has deemed potentially significant, which is the text as transformed into a paratext. Further, the algorithm itself becomes an argument for interpreting a text through a particular interpretive lens (i.e., the parameters of the algorithm itself), drawing connections that may or may not be clearly "present" or important in a traditional reading of the text. The significance of Ramsay's algorithmic criticism is that it affords scholars a new way of engaging texts for humanistic ends by algorithmically discovering meaning within a text. That said, 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 in addition to the original text.

CRITICAL CODE STUDIES: ALGORITHM AS COMMUNICATION

In contrast to Ramsay'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 that meaning is provided 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 all contribute clearly and explicitly to the interpretive experience. 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 and celebrated.

It is easy to consider the semiotic value of code when its syntax closely resembles that of natural language, and many languages use syntax resembling that of English. The idea that code could and should be written primarily for human readers, rather than for the computers that interpret the code, 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 properly or accurately by a computer was secondary. Essentially, the idea is that human collaborators should be able to comprehend any of their colleagues' work and the functional intent of that work. This perspective has been echoed by influential programmers since then, including in several texts on fundamental programming principles and practices (Kernighan and Plauger 1978; Kernighan and Pike 1999). Matsumoto (2007) urged his audience specifically to treat code "as an essay," and he demonstrated doing so through a series of example programs written in the Ruby language (477).

There is a relationship between saying (describing) and doing (computing or executing) in source code and the executable programs they describe, but this relationship is not always emphasized or addressed in specific code texts. "Codework," a kind of code poetry, blurs that relationship with a similar relationship in written language between "saying" and "appearing," what Lanham (1993) referred to as a bistable oscillation between looking "AT and then THROUGH" texts, an oscillation that is never eradicated but might fluctuate to varying extents between audiences and contexts (5). Cayley (2002) wrote codeworks in order to experiment with the possibilities of maximizing code's ability to perform computational tasks while also clearly communicating its goals to a reader in a conventionally understandable manner. For Cayley, "codework" as a term highlighted 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
```

(Cayley 2002)

The above was composed by Cayley in HyperTalk, a programming language developed in the late 1980s for the Macintosh hypermedia program HyperCard. Its syntax, like that of many other high-level programming languages, closely resembles that of English, making it easily readable by humans (assuming those humans understand English). The larger program from which this excerpt was taken is 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 the value of which changes depending upon certain conditions (such as whether the current value of **`media`** is greater than the current value of **`instantiation`**), not unlike the contingent meaning of *any* term for a particular context and discourse community. The meaning of the code, then, emerges not only from what the code text *says* in English (or at least near to it) but what it *does* computationally, even if we might view the functional purpose of this, or any other example, program to be trivial (Sample 2013). Cayley acknowledged the code's "ambiguous address" of both human reader and computer interpreter, implicitly suggesting that there existed rhetorical situations for each of these audiences (2002).

While Cayley's example is relatively readable and accessible, most code—as Marino (2006) has observed—does not closely resemble literature or other genres or forms of conventional discourse. This means that the dichotomy of audiences for code-based communication may *seem* to skew more toward the technological than to the human. A number of rhetoricians and critical code scholars have interrogated this balance, questioning the value of a traditional human-oriented communicative hierarchy in favor of a more distributed, relational network of human and nonhuman actors (Arns 2005; Cummings 2006; Brown 2015; Nicotra 2016).

Arguments in Code as Algorithmic Meaning Making

Just as algorithmic procedure has provided means for artistic and poetic invention, so too have algorithms served rhetorical invention, described and expressed not only in code but in a variety of other communicative

modes. The notion that an algorithm can and does work rhetorically is a radical departure from its conventional definition, which emphasizes discrete procedural execution of its components. 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 and its value, but those qualities do 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" (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. In other words, the scale of critical inquiry assumes that the point or goal of a procedure is already determined or understood, rather than remaining germane to the discussion about how best to solve a relevant problem.

The novice mentality described by Edmonds (2008) is closer to the mark for thinking critically in regards to algorithms and rhetoric than the conventional "learned" mentality, since the novice student has not yet been trained to disregard certain critical perspectives in order to approach the algorithm "correctly" for academic or industrial purposes. 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" (xvi, emphasis added). Berlinski's reference to a "prescription for action" is not just a means of defining a particular action to occur in a certain way but to call for that action to be undertaken by a certain audience, laying bare the algorithm as a procedural engagement with a rhetorical exigence.

What a recognition of algorithm *as meaning making* offers rhetoricians, then, is an opportunity to explore how a seemingly "machinic" manner of discourse—the code of digital technologies and media—can provide insight into the interplay among the canons of rhetoric to influence the potential expression(s) 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 corhetor (inventor, arranger, etc.) alter our understanding of the rhetorical 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 persuade one another with the aid of digital technologies, it is important to understand how we are capable of stimulating particular types of action through the use of those technologies. 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 from traditional approaches to rhetoric, which may take as given the technological mechanisms constraining particular discursive efforts.

Procedural Rhetoric

While it may appear trivial, there is a significant difference between algorithmic meaning making and the related idea of procedural rhetoric. Procedural rhetoric as described by Bogost (2007) deals with the influential qualities of procedure-based *systems* exerted upon individuals who make use of those systems. Bogost, however, did not elaborate on a wide variety of code-based algorithms as much as on how algorithms function in regards to interactive systematic procedures like video games and how games' procedures persuade players to act within a game's context. For Bogost and for others since (including Sample 2013), the rhetorical capabilities of a procedural system offered new ways of engaging specific populations that might otherwise be ignored or overlooked. 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) a game's developers intend for its content to be encountered by a player.

But 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 meth-

ods, 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, 3). This definition suggests that code—as a symbolic representation of algorithmic procedure—might play a prominent part of Bogost's discussion of procedural rhetoric. Bogost noted, however, that some processes that "might appear unexpressive, devoid of symbol manipulation, may actually found expression of a higher order" and explained how those humans who are perceived as "breaking procedure" in actuality are constructing new processes, and expressing them, in order to complete tasks (2007, 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.

Rhetorical procedure has been explored further by Lanham (2003), who referred to such procedures as "tacit persuasion patterns" (119). For Lanham, tacit persuasion occurs constantly, since we are almost never 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, 120). This description hints not just at a passive acceptance of rhetorical appeals but a subconscious engagement with the variables of a rhetorical situation as well as enthymematic arguments provided by a rhetor across multiple levels of language. As a scholar interested in speech and writing, Lanham focused primarily on persuasive techniques available in discursive 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. Such devices suggest a significance in the argument through the affordances of languages' stylistics. While Lanham did not address the possibilities of tacit persuasion patterns or devices in artificial (code) languages, they nevertheless exist and are already used frequently by many developers working collaboratively, as will be discussed in subsequent chapters.

Rhetorical code studies continues this conversation in regards to the potential for expressive action through the languages of computer code. Code might be described as "inexpressive" in that it creates and communicates meaning in ways that often differ from conventional invention and delivery of discursive arguments; it is precisely because of this quality, however, that its procedural nature can demonstrate expressive out-

comes in novel and unique ways. Here I set aside the specific expressive acts of procedural execution (which would highlight the general user's experience with a particular software program) and instead discuss 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 technological systems including user workstations, servers, network routers, sharing economies, etc. In other words, my interest in algorithms is focused on how code, as a medium to describe algorithmic procedures, is articulated for rhetorical purposes.

Algorithms We Live By: Recognizing Rhetorical Algorithms

In order to discuss how an algorithm can act rhetorically, I turn to investigate 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 necessarily 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, and Beck (2016) has demonstrated how algorithms and code function rhetorically through their performative and linguistic natures. One particularly significant quality of code, as with other forms of language, is the symbolic action (cf. Burke 1969) that a given code operation, or set of operations, provides, its descriptive qualities making its purpose and function(s) understandable to various human audiences and facilitating further activity.

Computational processes are similarly metaphorical; the rhetorical actions they symbolize succeed primarily because of the metaphor-driven ways human audiences are influenced to understand those processes as arguments that influence their audiences in particular contexts for particular purposes. It is also important to observe that writing or speaking about algorithms (or, potentially, even *to* algorithms; see Gallagher 2017) is itself a metaphorical activity that frames procedure as a kind of description. As Bogost (2007) has noted, "only procedural systems like computer software actually represent process with process" (14). With this in mind, I will identify some integral means by which code processes make meaning for developer audiences in what those processes do functionally and in how they are structured for the sake of human readability as well as for technical or technological expression.

In order to make this argument, I need to demonstrate how relatively

simple code algorithms appear to, and do, work rhetorically. In the following pages, I address several examples of increasing complexity whose code texts communicate at various levels the meaningful action they mean to effect through the computational operations that make up their text forms. First, I look at FizzBuzz, a program that iterates through a specific body of data, used in hiring tests to determine applicants' knowledge of algorithmic principles. Second, I examine quine, a program that outputs its entire code content. Third, I turn to HashMap concordance, a program that tracks word frequency across a set of input text (in this case, Mary Shelley's *Frankenstein* and Bram Stoker's *Dracula*).

CASE 1: FIZZBUZZ

The first example to be scrutinized, the "FizzBuzz test," is relatively simple in construction and intent (although elsewhere I have explored the rhetorical canon of style in greater depth than is offered in the discussion below; see Brock 2016). It is the focus of a common hiring test for programmers, intended to weed out applicants who are unable to work through the basic principles of algorithmic procedure and expression. The goal of this algorithm is to take the numbers 1 through 100 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 single line of text (although some variations on the test would include line breaks, spaces, or some other distinguishing markers between each output iteration):

```
12Fizz4BuzzFizz78FizzBuzz11Fizz1314FizzBuzz1617Fizz
  19BuzzFizz2223FizzBuzz26Fizz2829FizzBuzz3132Fizz
  34BuzzFizz3738FizzBuzz41Fizz4344FizzBuzz4647Fizz
  49BuzzFizz5253FizzBuzz56Fizz5859FizzBuzz6162Fizz
  64BuzzFizz6768FizzBuzz71Fizz7374FizzBuzz7677Fizz
  79BuzzFizz8283FizzBuzz86Fizz8889FizzBuzz9192Fizz
  94BuzzFizz9798FizzBuzz
```

Depending upon the language, and even more importantly depending upon *the way a programmer approaches the problem*, there may be dozens of

ways by which one could construct this algorithm effectively, as suggested by Lopes (2014) in regards to the task-frequency program she wrote in more than thirty different programming styles. The path ultimately taken by any particular developer to solve his or her problem provides a great deal of rhetorical insight about that developer's abilities, limitations, and preferences in computing data, using certain languages, and working in certain development environments.

While the immediate pragmatic goal of the FizzBuzz test is for an applicant to demonstrate to a potential employer his or her competence as a programmer, the applicant also demonstrates more generally an understanding of how "best" to use a given language (whether defined as computationally elegant, readable, or possessing some other quality). Further, the applicant also communicates through the written code how he or she *thinks* computationally through the frame of that particular language in order to solve certain types of problems. It is also important to note that my analysis of the FizzBuzz test, focusing on stylistic influence on procedural expression in code texts, is not meant to erase or elide the discriminatory tactics that can accompany use of such a test or other means of constructing organizational communities (Steinberg 2014; Burleigh 2015) or the underrepresentation of women and some minorities in the software industry and OSS communities (Lopez 2017; Reagle 2013). Both of these issues influence who is likely to have learned programming (and gain appropriate credentials or degrees) and pursue a position for which this test would be administered.

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. Example 2.2.a, on the left, frames its computation in an initial "catch-all" condition statement, that is, that `i` is a multiple of three *or* of five. (The syntax `i%3` checks whether "`i` divided by 3" has a remainder of zero.) Then it checks each of those subconditions independently of one another. This means 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"), without, in its current form, impacting the outcome of the other conditional computation. In comparison, example 2.2.b, 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, *only then* does the loop repeat 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 contains two examples of the algorithm as composed in the Ruby programming language. The major difference between these two examples written in Ruby lies not in how the specific conditions are constructed (although the construction does differ between the two) but rather 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. Example 2.3.a, on the left of table 2.3, makes use of a **for** loop, which—as with the JavaScript examples—iterates through a body of data and computes each item within that body before moving to the next item. In these loops, that data has been the set of integers from 1 to 100, but **for** is not limited to iterating numerical data. (In Ramsay's algorithmic reading of *The Waves* discussed earlier in this chapter, the novel's text served as the data population, separated out into units of individual words.) Example 2.3.b, 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. This particular example *incidentally* includes a

Table 2.2. Two example FizzBuzz loops in JavaScript

Line	Example 2.2.a	Example 2.2.b
1	`for(var i=1;i<=100;i++) {`	`for(var i=1;i<=100;i++) {`
2	`  if ((i%3==0) || (i%5==0)) {`	`  if (i%15==0) {`
3	`  if (i%3==0) {`	`  console.log("FizzBuzz");`
4	`  console.log("Fizz");`	`  } else if (i%3==0) {`
5	`  }`	`  console.log("Fizz");`
6	`  if (i%5==0) {`	`  } else if (i%5==0) {`
7	`  console.log("Buzz");`	`  console.log("Buzz");`
8	`  }`	`  } else {`
9	`  } else {`	`  console.log(i);`
10	`  console.log(i);`	`  }`
11	`  }`	`}`
12	`}`	

line wherein the value of **`i`** is changed; it is not inherently connected to the **`times`** method used here. Because of how **`times`** operates, counting begins at zero rather than one, and thus **`i`** needs to have an extra number added to its current value (in line 2, the first operation within the **`times`** block) before the remaining operations can accurately be computed for the FizzBuzz problem as it is posed.

Conceptually and metaphorically, this distinction between **`for`** and **`times`** reflects a fundamental distinction between iteration and repetition. The **`for`** loop suggests to human and machine readers 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 calculated and modified at any given moment. In contrast, the **`times`** "pseudo-loop" suggests that it will execute *the same operations* a set number of times, independent of any input variables; 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, more broadly, any block of code that processes a body of data—is a computational metonymy: the "loop," which implies a cyclical return to its origin, is actually 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 iterative execution transforms the code both in how it reads and in how it operates, just as the input data is transformed into appropriate output data.

While the function of the FizzBuzz algorithm, as represented by these examples, may not initially appear to be rhetorical in nature (since it checks a set of numbers and prints out numbers or words), it nonetheless

Table 2.3. Two example FizzBuzz loops in Ruby

Line	Example 2.3.a	Example 2.3.b
1	`for i in 1..100`	`100.times do |i|`
2	`  if i%3 == 0 then`	`  i = i+1`
3	`  print "Fizz"`	`  if i%15 == 0 then`
4	`  end`	`  print "FizzBuzz"`
5	`  if i%5 == 0 then`	`  elsif i%3 == 0 then`
6	`  print "Buzz"`	`  print "Fizz"`
7	`  End`	`  elsif i%5 == 0 then`
8	`  if i%3 != 0 && i%5 != 0 then`	`  print "Buzz"`
9	`  print i`	`  else`
10	`  End`	`  print i`
11	`End`	`  end`
12		`End`

serves as a concise example of the persuasive capabilities of code. Specifically, the FizzBuzz algorithm provides meaningful information to its applicant author, to any other human readers (e.g., the employer), and to the machine related to what the author understands about how to engage in the manipulation of a particular 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 how the author has identified at least *one* central means around and through which to compute relevant data and facilitate the desired result. Further, it implies a suggestion as to how one should understand and work with that data toward a perceived desired end.

CASE 2: QUINE

The second example to be discussed here is the *quine*, defined most concisely as a "self-reproducing" program. In other words, the output of a quine is the sum of its code content, meant to mirror that content perfectly. The term "quine" derives from the name of mid-twentieth-century logician Willard Van Orman Quine, who provided 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). Quine meant that the statement provided can accurately be neither true nor false, and it is only in the combination of concept and *quoted* concept that the paradox's meaning emerges. Rhetorically, the algorithmic quine offers an opportunity for a rhetor to consider what it means to construct a "logically sound" argument and how to communicate that logic to a given audience. Is the argument merely an appealing effort at constructing or suggesting the construction of meaning, or is there an internally valid consistency to a quine? Does the former require the latter? How transparent or opaque should its logical mechanisms be to the audience so that it can recognize (or not) how the rhetor works to persuade it to action? 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 in code, written in the Ruby programming language, is a single-line program provided in its entirety:

```
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 data type called a "string," a container for some arbitrary set of alphanumeric characters (whose boundaries are identified by the quotation marks preceding and following the string's content). The second operation in the quine, beginning directly after that same semicolon, recalls and displays for the user 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 quine program to be considered fully self-referential (i.e., the recall command itself has to appear in the output of the recall). When **`x`** is defined in part as **`%p`**, **`%p`** serves as a container in which other content might be substituted later; the statement **`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 noncomputing replication of those operations). This binary quality of the quine, for developers, allows them to make explicit note of what "runs" (i.e., what computes) and what is output (i.e., what is computed).

When we examine the code-based quine rhetorically, this distinction changes 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 as procedure and output; the action of the data is to highlight the means by which it was revealed, the computations constituting 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 an apparently complete persuasive entity. That is, because the quine outputs itself, "everything" is made clear to the user who executes it. It is this relationship that Cayley (2002) referred to when identifying 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 computational chiasmus that is completed when the input is ultimately displayed as output.

The quine *also* implies, though, that it is a simple program that merely presents a plainly stated message: 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 *make up* 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 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 concealing anything. This idea of language as apparently, but not actually, mimetic is key to an understanding of the quine and what it can achieve rhetorically (and, perhaps, reminiscent of the debate over technical communication as instrumental or humanistic; see Miller 1979; Johnson 1998; Moore 1999; Johnson 1999; and discussed further in Dubinsky 2004). 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 and that no further critical inquiry as to its purpose or mechanical procedure(s) is necessary.

One complication for the quine—and thus, by association, any executable software program—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 UNIX 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. Whenever one interacts with an external entity (e.g., code written by another person), it may not only be *difficult* to tell whether some meaning is concealed but it may be *impossible* to verify whether any concealment has ever occurred unless attention is drawn thereto (cf. Miller 2010b). 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, in what Chun (2011) has described as "a crafty, speculative manner in which meaning and action are both created" (24). Just as with more conventional rhetorical activities that focus 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. This argument calls attention to the ability of code to do more than it suggests, especially when it suggests that one has access to the code in its entirety, including the full functionality of its expressive ability.

CASE 3: HASHMAP

The third example to be discussed transforms and deforms existing texts—and conventional readings thereof—in order to bring to light new meaning that might not have been exposed in the text's original format. This sort of code, and its paratextual output, is aligned closely with the algorithmic criticism of literature promoted by Ramsay (2011) and others. 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 to engage different audiences rhetorically. This example was composed by Shiffman (2014) for, and to be included with, the Processing integrated development environment (IDE), which 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 so that each has its own identifying **`key`** data. By default, the program takes a plain text file with the contents of Mary Shelley's *Frankenstein* and Bram Stoker's *Dracula*, the text for each novel made available through the electronic Project Gutenberg public domain library, and displays each word in a particular font size related to the frequency of each word's occurrence in

either novel. The program will *not* count a word if it appears in both novels or if it appears fewer than five times. The novel's words, reconfigured as a **HashMap** (a kind of data table that stores paired "key" and "value" data), offer in their recombined form a way to read meaning in each text, and in comparison to the other text, based upon the frequency of particular terms and concepts. Table 2.4 contains excerpts from Shiffman's code, which appears as an example project in two files ("HashMapClass.pde" and "Word.pde").

The program takes the contents of *Dracula* and *Frankenstein* and strips out all punctuation so that only words (data "strings" of characters, referred to as hash map "keys"), and the spaces between them, remain. Then each word is checked against the current contents of the **HashMap**

Table 2.4. Excerpted HashMap example code by Shiffman (2014) written for Processing, from "HashMapClass.pde"

Line	Code from "HashMapClass.pde" file in HashMapClass example (Shiffman 2014)
27	`  words = new HashMap<String, Word>();`
[...]	[... *These lines create variables populated by the novels' texts*]
50	`void loadFile(String filename) {`
51	`  String[] lines = loadStrings(filename);`
52	`  String allText = join(lines, " ").toLowerCase();`
53	`  String[] tokens = splitTokens(allText, " ,.?!:;[]-\"'");`
54	
55	`  for (String s : tokens) {`
56	`    // Is the word in the HashMap`
57	`    if (words.containsKey(s)) {`
[...]	[... *These lines locate the appropriate key and update its value*]
68	`    }`
69	`    else {`
70	`      // Otherwise make a new word`
71	`      Word w = new Word(s);`
72	`      // And add to the HashMap put() takes two arguments, "key" and "value"`
73	`      // The key for us is the String and the value is the Word object`
74	`      words.put(s, w);`
75	`      if (filename.contains("dracula")) {`
76	`        w.incrementDracula();`
77	`      } else if (filename.contains("frankenstein")) {`
78	`        w.incrementFranken();`
79	`      }`
80	`    }`
81	`  }`
82	`}`

and, if it does not currently hold a position within the **`HashMap`** container, is added thereto with a value of 1. If the word *does* already exist within the **`HashMap`**, then the appropriate value for that word is increased by 1. Once this check is completed, it is repeated for the next word in the novel, and then the next, until the entire novel has been iteratively "read" in this fashion. This activity—or at least an activity closely related to it—is described as "text mining," referring to the act of extracting meaningful data from an otherwise opaque or unexamined source text.

As these word count checks occur, each word that has been computed is written onto the screen at a random spot along the top with its incrementer value influencing the size of that word on the screen. Words that occur more frequently in either novel are displayed in larger font size and move down the screen more quickly, and only so many words are displayed on the screen at any given time, even though the total calculation of the novels' contents continues to execute. In other words, there is a distinction between what the user and his or her machine experiences, with the former engaged in a particular constrained reading of *Dracula* and *Frankenstein* that is acutely distinct from Processing's interpretation of the data as solely important in its numerical sense.

There are several meaningful activities that occur within the lines of the **`HashMap`** code that, together, dramatically alter the traditional concept of reading and writing. Perhaps the most significant of these is the incorporation of object-oriented programming (OOP) principles into these activities. In OOP, distinct "objects"—clusters of expressive code entities—are created from a framework of rules (called a "class"); each instanced iteration of an object obeys the same procedural rules but responds to those rules independently of other object instances; that is, it calculates and expresses its procedures without any *inherent* influence on any other object, although this may occur incidentally. The result is a set of objects whose behavior has the same fundamental principles but that emerges uniquely for each individual based upon the constraints of its being called or created as part of a given program; in this example program, each displayed word is a separate object. (As an aside, OOP is unrelated to the philosophical subfield that has been named object-oriented ontology for its focus on nonhuman entities.) What OOP allows for is the potential for a 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

Fig. 2.1. Example expression of Shiffman's (2014) HashMap code

is slightly flattened as it is expressed within the constraints of a particular situation; for example, as the **`Word`** "Lucy" is counted by Shiffman's program, the meaning of each use of the name reference 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`** object defined by the code, becomes notable primarily as a unique string of alphabetic characters that repeats so many times over the course of the novel, represented visually in contrast to other word strings by its expressed size on, and speed with which it travels across, the screen. Lucy—along with any other word from either novel transformed by this program—gains significance through certain demonstrations of numerical frequency. Because of the way Shiffman coded his program, there are a few intriguing side effects of this shift in significance (see figure 2.1).

Through the expression of Shiffman's code, it becomes clear that the most frequent words in *Dracula* (and, potentially, in most novels or other substantial bodies of text) are words that might be considered the least important: the conjunctions, articles, and prepositions that link together more "significant" concepts and messages. It is possible to add into the code exclusionary conditions that would ignore these seemingly trivial words (often referred to as "stop" words), such as the single character "t" in figure 2.2, likely a remnant from the splitting of words by punctuation characters, outlined in line 53 of table 2.4; to support this theory, the word "didn" appears below "t" at roughly the vertical midpoint of the figure, nearly blending into "flies" to its right. Any subsequent analysis of word prominence via frequency made while ignoring such words or word fragments could be considered technically inaccurate or incomplete. Further, the decision to reveal the computations of word frequency in this manner

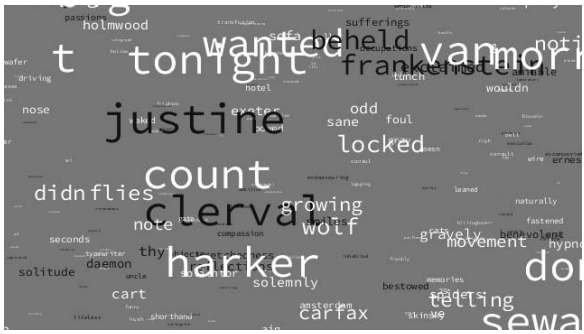


Fig. 2.2. Later expression of Shiffman's (2014) HashMap code

results in a very different visible outcome than through some other approach, such as generating a text list of each word and its final frequency count. Because only a subset of words from each novel appears on the screen at any moment, the user ultimately is left unaware of the significance of words that do not continually appear or that do so in a small font size (see figure 2.2).

In figure 2.2, a screenshot taken at another moment during the same expression of the code displayed in figure 2.1, there is a noticeable difference in the makeup of visible words. While the code continues to cycle through the words in each novel, the user is only given access at any time to a small set of the computational results. Words such as "mina," "murderer," and "bag" (visible in figure 2.1) have all been repositioned beyond the bounds of the program's output window, obscuring the visualizations of their frequency; but "harker" has cycled across the screen again at this point, its position unintentionally captured at nearly the same vertical point as in figure 2.1.

Why might a program seemingly so trivial—at least in terms of how it presents a "reading" of *Dracula* and *Frankenstein*—be worthy of discussion? The choices Shiffman made in creating this program signal to user and to developer a particular set of values related to reading and (re)writing these novels algorithmically. 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 expressive arrangement/layout. At the same time, the way Shiffman has chosen to determine what defines 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 or she inputs. This is not simple instruction entirely 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 if we can understand how we are expected to interpret that argument and read the created paratext in a particular way.

Conclusions

More generally, each of these examples discussed above serves to demonstrate some fundamental qualities of code as a significant form of making meaning rhetorically. While on its own each program may seem to serve only a limited purpose, together the code functions provided in these examples work—alongside thousands of others used daily—in more robust software programs to persuade and influence numerous human and technological audiences to act in rhetorically meaningful ways. These ways may not always be visible, clearly recognizable, or discursive in nature, but they nonetheless create meaning and work to persuade the human and nonhuman audiences they engage to induce various types of change in the ecologies in which they operate.

Further, the potential for meaningful symbolic action that is demonstrated in these examples reflects the close connection that has existed for millennia between algorithmic procedure and humanistic activity, in terms of both creatively and critically oriented work alike. While the vast majority of scholarly 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 demonstrated through digital and computational media—as means of persuasion is relatively sparse. Rhetorical code studies, however, 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 in audiences.

The following chapter demonstrates the potential of code to facilitate action through an examination of the spheres of discourse *surrounding* the composition of code in development communities. Exploring written communication about code can provide a means of demonstrating the rhetorical strategies used by programmers to develop and promote software among certain populations (namely, other current and potential pro-

grammers). The values that developers stress in their code and discursive commentary, in addition to the persuasive tactics they use to build particular types of identities and communities, highlight a set of qualities likely to be communicated in and through the code they produce. Building on this foundation, we will then turn to bodies of code texts to see how those persuasive expectations—and the strategies believed to facilitate the realization of those expectations—play out in a given development case.

CHAPTER 3

"I Have No Damn Idea Why This Is So Convoluted"

Analyzing Arguments Surrounding Code

> Decades of efforts have gone into helping civilians write code as they might use a calculator or write an e-mail. Nothing yet has done away with *developers, developers, developers, developers*.
>
> –Paul Ford, "What Is Code?"

Despite the long-standing relationship that exists between algorithmic procedure and humanistic activity, it has only been relatively recently that rhetoricians have investigated how algorithms function rhetorically (and how they are *understood* to do so) for computational and communicative purposes. This set of qualities is key: while algorithmic procedures perform meaningfully, those who compose and execute such procedures may not always recognize how those procedures serve them in a rhetorical sense. Thus if we are ultimately to examine how code is a means of constructing meaning via algorithms, it is first necessary to investigate how those involved in the development of software programs perceive algorithms as rhetorically powerful. This particular investigation can best be performed through the body of discursive communication written by programmers, professional and amateur alike, for other programmers about how their code does or should operate.

Programming discourse assumes a prominent role in this chapter specifically because it is clearly rhetorical in the most conventional sense, in that developers persuade one another to accept the choices each has made in developing meaning through particular code texts. Each developer often takes a position where they must defend the rhetorical choices made in composing a particular piece of the overall software program under dis-

cussion, and the arguments made in that defense bring to light a number of values possessed by that developer regarding: 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.

There are several genres in which programmers predominantly communicate with explicitly rhetorical goals. Email (individual messages and electronic mailing lists), discussion forum posts, and chat environments—such as IRC and, more recently, Slack—provide opportunities for developers to engage in sustained synchronous and asynchronous conversation with their peers about code: solving particular computational problems or bugs, deliberating over preferred software and hardware architecture, and even addressing perceived differences between developer and user knowledge about a program and its use(s). More "official" documentation genres, provided as code comments and related files (e.g., "Readme" text documents) outlining the intended purpose of a given program, offer programmers a means of articulating a range of formal and informal suggestions for how others adopting or modifying a given program's code should or should not approach doing so. Similarly, versioning systems serve collaborators as arenas for deliberation on specific code texts' relevance to and strength or value in improving a particular program through the acceptance or rejection of specific changes to its code base.

To begin investigating the role(s) of programmer discourse on rhetorical communication *in* code, I relate discursive activity on code to traditional discourse studied by rhetoricians. The rhetorical qualities of such communication can tell us a great deal about the considerations that developers weigh when it comes to discussing their work with collaborators and other audiences. While this sort of discursive activity only partially describes the range of meaning communicated by developers to development audiences, it nonetheless demonstrates the kinds of meaning potentially created and communicated *in* code as well.

Rhetorical Scholarship on Online Discourse Communities

Because much of the communication that occurs among software developers happens online or in networked electronic environments (e.g., within a given team, among departments within an organization, or across geographically distributed communities), we can examine rhe-

torically the bodies of texts composed for their diverse goals. 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 tends to examine online communities as electronically transplanted iterations of conventional communities, with their forums and email lists as sites for gathering and exchange that are perceived to function as digital iterations of historical *agorae*. When examined through this lens, digital environments are understood not necessarily to possess qualities unique or otherwise distinct from those of more traditional communities or forms of communication. Instead, groups communicating digitally are examined as extensions of traditional discourse communities that employ different, but historically relatable, media from their predecessors. The second approach to rhetorical inquiry into online communities involves exploring those communities as fundamentally distinct from conventional groups with similarly distinct means of (and preferences for) communication among group members. In this approach, online communities are generally positioned as networks with unique qualities that would be impossible to replicate—or otherwise have no clear or familiar analogue or precedent—in nondigital environments.

While these two approaches differ 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. Both approaches are complicated by the integration of code into rhetorical examinations 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. Should it be understood as an electronic transcription or inscription of Boolean logic (which does not need digital technology to operate in the most technical sense)? Or is it a form of communication realized in full only through the capabilities of electronic, networked computer systems? The likeliest answer is that it is both of these and more.

Admittedly, many of the studies that viewed online communities as digital extensions of conventional communities are dated, with the majority of such publications spanning the late 1990s or early 2000s. In part, a number of scholars may have focused on conventional qualities of online communities due to the limitations or unanticipated uses of Internet spaces, such as Benson's (1996) examination of Usenet groups' digital extension of political discourse or Silver's (2005) discussion of electronic

shopping centers as town halls. Gurak (1997) provided an extensive examination of a variety of online community responses to two failed technological initiatives, Lotus Marketplace (a marketing database program) and the Clipper chip (an encryption device to be overseen by the US government). Gurak's analyses of online postings about these controversies ultimately framed online discussions as hybrid forms of conventionally recognizable speech or writing, possessing both the immediacy of speech and the permanence of writing.

More recent studies qualified these earlier arguments in regards to the specific attributes of electronic communication. Warnick (2007) highlighted the potential problems of access, literacy, and clear author identity surrounding digital technologies, each of which might hinder an otherwise more complete online reconstruction or iteration of more conventional public discourse. Hocks (2003) 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. Kimme Hea (2007) observed the difficulties that researchers must often navigate when studying online communication, thanks in part to "the Web's mutable nature; continued growth; and confluence of visual, aural, and hypertextual forms" (270). Miller and Shepherd (2004, 2009) contended with the complexities of studying genre and platform in an era of rapid technological advancement and change. Shared among these scholars' arguments is the sentiment that while there may be new contexts or ways of understanding how online communication works that need to be incorporated into scholarly consideration, there may also be productive continuations of convention or tradition that should not be overlooked.

A number of scholars have advocated an alternative approach to theorizing online community rhetoric, one in which digital environments and rhetorical situations are considered fundamentally distinct from those surrounding conventional communities. For example, Brooke (2009) called for a new media reconsideration of the rhetorical canons and of the trivium of classical education, with a focus on how new media demand innovative rhetorical perspectives if we are to understand how we communicate in ways unanticipated in previous contexts. Shirky (2009) observed the contributions of self-organizing communities to construct meaningful and dynamically shifting arguments through projects like Wikipedia, whose developments can be traced through the version histories of particular subject-oriented pages. Goodling (2015),

observing digital forums as a "new public sphere," identified essential qualities of online activism and engagement that were simply impossible to achieve through other means, from real-time citizen journalism to born-digital versions of civil resistance. Gallagher (2017) pointed toward a growing need to attend to algorithmic agents as audiences for digital and online communication.

The question that drives research relating to online communication as different from conventional communication is: what, if anything, makes such discourse *unique* or innovative? Answers to this question, whether historically conventional or otherwise, are significant because they help explain the potential range of (1) interactions among, (2) purposes of, and (3) rhetorical means available to the members of various online communities. While these methods provide crucial insight into the ways communities' members interact, there has been little scrutiny of how online communities' members make and share meaning with one another through the code that makes up a significant part of their discursive interactions. The ability to communicate in such ways 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. We can see the complexity of these means of communication in an examination 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.

The Rhetorical and Social Makeup of Open Source 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, and intensity of dedication or enthusiasm among their 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 interact with one another.

This significance of technology as an 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 Initiative, 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 temporary if not outright ad hoc in nature. Further, its members are likely to be dispersed across the world thanks to increasing 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 theoretically 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 access profoundly blurs the boundary between the conventional labels of "producer" and "consumer" and opens up the potential for a hybrid identity—what has been dubbed the *prosumer* (Ritzer and Jurgenson 2010)—as both developer and user of a given software program. Such a sense of access and productive capability is central to the success of many OSS projects; many can count themselves as members of communities that they might have otherwise been excluded from (or considered themselves excluded from). For example, the website for the Red Hat Fedora operating system's community referred to its members as "friends" that contribute, in different ways, to the operating system's success (Red Hat 2017). Because of this shift in perception, the dynamics of those communities—and thus the ways their members communicate—change dramatically; the variety of purposes members have for participating therein increases, including whether members choose to participate

through improving on the program's code or discussing ways in which it might be improved.

As a result, what makes many software development communities stand out from 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 such development communities can attempt to communicate with 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 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 communication but instead to observe how rhetorical 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.

It is also worth observing that these communities do not exist in social or cultural vacuums and neither should they be implicitly understood to reflect the full diversity of a population. According to a GitHub (2017) survey of roughly six thousand contributors to OSS projects, 95 percent of respondents were men, while 3 percent were women and 1 percent identified as non-binary (n.p.). The same survey revealed that 26 percent were immigrants "to and from anywhere in the world," and 16 percent were an ethnic or national minority in their country (GitHub 2017, n.p.). Seven percent of respondents to the GitHub survey identified as "lesbian, gay, bisexual, asexual or another minority sexual orientation" and 1 percent were transgender, with this last group making up 9 percent of the respondent group of women contributing to OSS projects (2017, n.p.). Clearly, there are certain backgrounds and experiences that, as the survey has suggested, are more likely to be represented than others in OSS development, and their influences can frequently be recognized not only by scholars but also by other developers and users as well.

Common Structures of OSS Development Communities

For more than twenty years, OSS has gained ground in the software market as an alternative model to the traditional system of proprietary pro-

duction and distribution. The defining qualities of OSS include free access to source code and communal development and improvement of software programs, "free" meaning unlimited or unhindered, but in many cases also referring to a lack of financial cost; "libre" has also been used for the former meaning so as to distinguish it from the latter. Individuals possess the legal right and, for *some*, the knowledge to experiment with and make adjustments to OSS program code. 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 in new iterations of the program, theoretically *ad infinitum*. Among the most notable OSS projects include the Linux operating system (of which dozens of distinct distributions exist, including Red Hat's Fedora, Debian, Slackware, Ubuntu, and Gentoo), the Mozilla Firefox web browser, and the Apache web server. While these few programs listed here are among the most well-known OSS projects, their popularity nonetheless demonstrates 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 inherently or even *radically* different in terms of development structure than its traditional proprietary counterpart. That is, in both paradigms, there is generally a community of dedicated developers working on an active software program and a community of nondeveloper users that may or may not contribute suggestions for improvement (such as through bug reports or requests for features). Where OSS tends to differ from proprietary development is that there is often some overlap between "developer" and "user" (i.e., the prosumer role, cf. Ritzer and Jurgenson 2010), both in the amount of contributed aid provided by developers or users and the amount of perceived importance of any community member.

In many OSS communities, 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 and 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 and Howison 2005, 2006; Wiggins, Howison, and Crowston 2008). These studies, however, generally suggest that a clear organizational structure, rather than a dynamic and chaotic environment, has been demonstrated to be the most effective means of ensuring project

success. In fact, Crowston and Howison (2006) suggested that the optimal structure of an OSS development community "is onion-shaped, with distinct roles for developers, leaders, and users" (114). This onion has at its center the primary source of social power and influence, made up of the project founder(s) and the release maintainer(s). The further from the center each layer is, the less influential and less actively involved in the direction and improvement of the project it is, with the outermost layer consisting of those passive users who do not contribute at all.

While Crowston and Howison's onion metaphor is helpful in describing the general hierarchies constructed for a given software project, it obscures the dynamic movement that can occur across these layers and privileges (or responsibilities). For example, an individual overseeing a significant section of an OSS project might be near the center of the onion of that section, but that individual may be less central to the development of another major section of the project or to development of the project as a whole. As a result, it may well be more useful to think of the onion as having intersecting, Venn diagram–like networked links that cross at multiple node points, rather than layers that engulf others.

RHETORICAL INFLUENCES ON OSS DEVELOPMENT SOCIAL STRUCTURES

If an OSS community can be conceived of as a complex onion structure with intersecting and hierarchical layers, then how can we comprehend the rhetorical activity that occurs throughout it? How generalizable are such observations regarding communities with different social dynamics? This is admittedly a difficult act of inquiry to engage in, given the fluid nature of many OSS groups; the "onion" metaphor used by Crowston and Howison (2006) applies to a particular moment in time and is employed for their use in regard to a specific perspective (i.e., the overall development of a project). Unfortunately, the onion metaphor fails to take into account the continued meaningful communication that occurs among developers as they work to influence one another on the specific contributions they offer to part or all of the community, just as it describes only a subset of the makeup of software development communities and workflows that they engage in, OSS and otherwise.

That said, the onion metaphor can be incredibly useful when it comes to emphasizing the strata of valued roles and contributions to a project; like the translucent quality of any individual onion layer, the impact of individual contributions on a given project or its development community

can be seen by looking "through" or "across" the project's layers (e.g., its social circles or the functional capacities of its code). By describing the structure of an OSS community via the metaphor of an onion and its many layers, Crowston and Howison (2006) suggest both that there is an inherent structure (with organizational power increasing as one moves toward more inner layers) *and* that all the layers are similar. This suggested similarity is significant for a generalized understanding of 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 past several decades, even as they ultimately stressed the benefit of such work (McNenny and Roen 1992).

Not surprisingly, the potential for rhetorically effective communication among the members of a given community is greatly affected by the structure of that community. The organizational metaphor of an OSS community as an onion, whether envisioned with isolated or intersecting layers, suggests that there are increasing levels of social significance and celebrity as one moves toward its center (with outer layers orbiting the inner layers), as these qualities are—theoretically if not actually—both more valued by that community. In this sense, the community appears to function like a more traditional discourse community or social entity, such as having one or more central figures (e.g., a president) make large-scale executive pronouncements that affect the entire community, while individuals in less powerful positions appeal to community leaders or intermediaries for change.

But—as has been noted by numerous rhetoricians from Gurak (1997) to Moxley (2008)—electronic technologies have complicated the influence and persuasive power of a community's various members, and the diverse needs of a community may catalyze intriguing intersections of members that the onion metaphor does not accurately address. Often, this complication exists as a form of hierarchical flattening. 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. A community's members gain the *potential* to engage social and organizational spheres (if not the entire community) in ways they might not traditionally have had. With any number of motivations,

such as a need to maintain enthusiasm among developers (or possible developers) in order to keep a volunteer-maintained project active, as well as a desire to communicate information about specific individual or group activities in order to coordinate specific or overall project functionality, many OSS communities seem to be composed less of discrete "onion" layers than of complex intersections of spheres of influence, only some of which may be hierarchical; even then, the hierarchy may be temporary as often as it is permanent.

Among the OSS development communities that have been notably affected by the "flattening" and other complicating qualities of electronic technologies upon discursive practices is Red Hat's Fedora Project. Their website described the Fedora Linux OS as a program built by a community that stresses a close relationship among its members: "The Fedora community includes thousands of individuals with different views and approaches, but together we share some common values. We call these the 'Four Foundations': Freedom, *Friends*, Features, and First" (Red Hat 2017, 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 a 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? Does it resemble OSS dynamics, broadly speaking? That is, do contributors to an OSS project actually possess such a bond with their colleagues, or is it a marketing tool meant to entice more potential developers into the project community?

Admittedly, it is incredibly difficult, if not outright impossible, to predict precisely how any given OSS development community determines the social standing of its members. In many cases, developers solicit contributions 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 a person entry into a valuable community but also affords status therein. For the example of Fedora and its community of "friends," it's not only important that people use Fedora, but that they perceive their relationship with other contributors in a particular way so as to collaborate more effectively on the project. Further, this relationship is explicitly recognized and endorsed by the overseers

of the project itself, since it is purposefully advertised as one of the core strengths of the Fedora project.

Of course, it is rare that any OSS project articulates how it rates or otherwise evaluates its members in any regard other than the merit of each individual's contributions to the project. After all, arguably any other criterion would suggest that development itself is not the most significant or central reason for development-related recruitment. "Development" as a broad descriptor of activity, however, has increasingly been described in more inclusive terms in order to bring in a wider range of potential contributors who may not necessarily be interested in writing code. 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 2018). Clearly, not all the proposed contributions demand the same skills or time of its developers (or even attract the same individuals), and coding itself is considered only one aspect of a larger set of valuable activities related to the project. 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 or those uninterested in writing code for GIMP.

But the existence of inclusive calls like those from the GIMP development community is not necessarily a reflection of a generally inclusive mindset. That is, not all contributions are viewed equally by all developers in all communities. Given the semihierarchical nature of many projects (whether onion-like in makeup or not), it is much more likely that particular contributions—and particular *types* of contributions—are generally valued more highly than others (Stewart 2005). 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, usually due to the "direct" influence of writing code on the project; new (or revised) code has a clear and immediately functional impact on use of the program. Linus Torvalds, the inventor of Linux, is a particularly prominent example of this contribution mindset and the paradox it can create: his previous contributions to numerous versions of the operating system's code has enabled him to retain a semiofficial role as "benevolent dictator" of Linux kernel development from its outset even

though he is not currently 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).

Effectively, Torvalds directs the main development of Linux via his establishment of a throng of "lieutenants" likely to promote code that seems to work in alignment with his vision for the operating system and to reject code that does not. When those outside his trusted circle propose changes to the Linux code, especially changes that may not align closely with Torvalds' philosophy on what Linux should do, Torvalds has been known to react passionately, sometimes due more to the mode of submission for a proposed contribution than to the merits of the contribution's code content (with one such example discussed in more detail later in this chapter). In other words, Torvalds can significantly influence the trajectory of Linux development due to the ethos he has built over two decades from his programming work on the operating system. His current "indirect" involvement in the project, however, can still overshadow the direct contributions of others who lack strong enough community standing to have an equal voice in the conversation. As a result, code and coding knowledge clearly matters, but it does so as a component of a more complex system of social influence in which other community members' perception of one's knowledge also plays an important role.

In other projects, positions of authority and leadership may be determined in a wide variety of methods. In some communities, leadership roles may be passed from one individual to another in appointed succession or by selection of the "upper management" circles of a development effort. Still others, like the community for 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, nominees highlighted the work they felt best suited them for the position of project leader, and each nominee's perception of significant or valued work is often substantially different from that of others (which, of course, suggests a number of different potential directions that the overall project might go). As a result, at each election the community is provided with an opportunity to consider multiple potential directions of the Debian project, some of which might otherwise never have been considered or discussed at length or at all.

While there are many development communities that have similar gen-

eral structures and philosophies—and in some cases shared members—each is *ultimately* unique in its social makeup and project mission. The specific goals that drive development activities in relation to one project are not uniformly possessed by, or valued in, another project. Stewart (2005) observed that social status in OSS development communities was driven in large part by communicative efforts as much as by any potentially "objective" or empirical evaluation of merit (e.g., the consistent demonstration of writing code concisely, for computational efficiency, or in some other manner valued by a community). Stewart noted that available social references—which may or may not have correlated with explicit knowledge of others' skill at programming—informed much of the determination of specific developers' social standings among their peers and colleagues. In fact, those parties interested in social status who established themselves quickly in a community—as willing to participate in development activities 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). Development activity and success is thus impacted significantly by *kairos*, a grasp of time, place, circumstance, and audience that enables a rhetor to effect change.

Indeed, *kairos* is an integral component of social capital, an individual's capacity to contribute significantly and meaningfully to a particular community, Those who recognize and make use of opportunities to position themselves as valuable are often those who ultimately gain some sense of authority over the community's central project, even if their long-term contributions do not end up as central or powerful as those of the more low-key or "latent" contributors considered to be less important to the overall community. These other contributors may, depending upon the nature of the project, spin off or "fork" their modifications to a different, albeit related, project where they *can* gain the social standing or authority over the program that they lacked in the original community. That is, we may view a given act as kairotically both successful and unsuccessful for the impacted communities (the existing initial project and the newly created fork project).

According to Nakakoji, Yamada, and Giaccardi (2005), the group-oriented goals of each development community are the adhesives that maintain connections between members and other projects. Just as it can be impossible to determine how long a given project might survive (be-

cause of changes in market demand due to perceived need thereof or the amount of competing projects that exist), it can also be incredibly difficult to determine how long that project's developers will remain active within the project community. 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, and Crowston (2006) stated that the makeup and mission of a project is less likely to change at its center than at its periphery; the overall range and makeup of participants—and thus the range of motivations involved in community participation—can be quite dynamic and varied. Some developers may contribute in order to improve a given program, while others may contribute because that program impacts use (or development) of a related program, while still others may contribute because the program is central to their employment-related activities and thus program development facilitates those other activities.

This motivational force is in line with the concepts of both *kairos* and rhetorical agency: parties involved in a particular rhetorical event possess a temporal and situational energy that facilitates their engagement with one another (Miller 2007). Agency, understood as this energy, extends in the context of software development to describe the meaningful and rhetorical action of large- and small-scale program development (i.e., contributing broadly to the development of a program and specifically to the development of one or more features or components thereof). In many cases, relevant rhetorical motivation is successful specifically because it imitates in various ways the energy of a productive business environment. That is, the model of a development community is often understood most clearly as a kind of corporate or commercial operation, regardless of whether that community is actively pursuing a profitable outcome for their activities or is organized remotely like a business. Ballentine (2009) has discussed the complications involved in opening or obscuring source code in order for a project to compete in particular types of markets as well as in regards to how such decisions affect subsequent work in those markets. That is, developers may—consciously or otherwise—contribute to a project as much to package and distribute the relevant product for industrial ends (e.g., profit, market dominance, industrial prestige) as they do to collaborate or to promote OSS principles. This is due in many cases to OSS "volunteers" working on specific projects because, as noted above, those projects are valuable to the volunteers or their employers. In some of

these cases employers provide a set number of compensated hours each week for those employees to work on OSS projects that might have some direct or indirect benefit to the employer organization.

OSS development is thus an incredibly 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. That is, it is possible—and often profitable—to build fences and walls around the "free" software supposedly accessible to any interested party. The agency of OSS freedom communicated to would-be developers can also be co-opted into closed source, proprietary agency denied to any parties not explicitly involved with those corporations.

In other words, the rhetorical power generated by OSS communities through common OSS collaboration and distribution models is, in many circumstances, also sapped or circumvented by proprietary competitors' ability to outmaneuver it for specific business-oriented ends. For OSS communities to function most effectively in achieving various forms of success—e.g., a sustainable funding model, a robust population of volunteer developers, widespread usage of the program under development—the rhetorical awareness of their members to engage one another in productive and collaborative communication is necessary. But it is not always clear what options any individual member has, what sort of choice that member makes about how to engage others, or why that member makes a given choice. Further, the kind of meaning communicated—whether in regard to individual or shared community goals—is similarly and frequently obscured. It is precisely in this obscurity, however, that we have an opportunity to gain significant knowledge about the means by which developers engage in rhetorical meaning making with one another and with other audiences. In this way, we gain access to understanding about how programmers work rhetorically, not only in their discourse about code but through their code as well.

Common Rhetorical Aims of OSS Developers and Communities

While each OSS community has its own unique set of goals, many individual aims overlap, sometimes significantly, with those of other communities, even if those aims are not explicitly communicated in a community's governing or design documents. Perhaps the most universally

shared goal is one of continued existence: most developers connected to a project hope for that project's development to continue indefinitely, even if they are not associated with the project indefinitely. Accordingly, accessible information about the project's mission and details about valued forms of contribution are often necessities for volunteer-staffed projects, as individuals may have less dedication to those efforts that provide little tangible benefit or social cachet.

This general goal, however, does not fully articulate the diversity of purposes toward which various individual and collective members of a development community operate. For some, there is a desire to steer the development of a program in a certain direction, such as to conform to a community's "best practices" or to facilitate one's own vision. Others may demonstrate their programming skills in order to bolster their reputation or to find employment. Others still may simply want to experiment with code or particular functionalities and share the products of their experimentation with their colleagues and others. Even this list is not exhaustive, but it serves as a starting point for investigation; in the following pages, I explore brief examples of rhetorical efforts relating to these goals so as to illuminate directions for future relevant research.

Perhaps the most commonly shared goal, as noted above, is the increased distribution and use of a particular OSS program. But there are also developers who are uninterested in promoting a program to broader communities or in ever relinquishing control over the program's development trajectory. Given that the success of many OSS projects is reliant upon a regular stream of new participants to replace inactive contributors, though, it should not be surprising that "survival" becomes a driving factor. In almost all cases, the appeal to survival and prosperity is intertwined with developers' attention to *kairos*. The Mozilla Firefox browser—which will be explored in greater detail 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 functionally and aesthetically preferable alternative to Internet Explorer, the default browser packaged with Microsoft Windows. Hundreds of developers flocked to the Mozilla project because of some discontent 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. But the development of Mozilla—like that of many other OSS projects—has

not been without rhetorical tension, often centered on the standardization of code texts within the frame of an individual organization or project.

STANDARDIZATION PRACTICES, KAIROS, AND COMMUNITY UPTAKE

Many OSS developer goals are relatively nuanced and less ubiquitous than those of large-scale institutions, with individual motivations and stylistic approaches to code impacting the work activities in which programmers engage themselves. Some developers seek to align their code practices with individual project style guidelines (and, in some cases, in adjusting those guidelines to more closely resemble existing individual code practices). This sort of effort *can* improve code readability across files as a move toward consistent style and logic helps readers learn to expect particular approaches to problem solving and the like. Such an effort can, however, also potentially inhibit other developers' contributions if those others do not follow the project's guidelines. Alternatively, standardization efforts may *rework* contributions to effectively remove the individual character of its author from the file, making each OSS contribution an effort to balance the preferences and needs of the community with the preferences and abilities of the individual contributor.

Concerns of standardization and normalization are key to a critical and rhetorical understanding of code. That is, the processes by which algorithms are composed 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 a given community has regarding its product(s) and the functions thereof. This is not to suggest that there are no circumstances in which particular operations or functional patterns are objectively superior means of achieving particular goals. Instead, I call attention to the persuasive influence of the computational and rhetorical *methods* by which developers pursue those goals and by which they deliberate their work. Just as scholars including Bogost (2007), Brown (2015), and Beck (2016) have explored procedural rhetoric relating to computational systems and algorithmic logic, so too can we consider here the discourse surrounding procedural composition, discourse ultimately meant to facilitate action as described through those procedures.

One such discussion took place among França et al. (2012) in regards to a proposed change to Ruby on Rails (often referred to simply as Rails), a popular OSS web application framework (examples of well-known Rails-based sites include Twitter, Groupon, and Hulu); it works as a ser-

vice that makes use of the Ruby programming language to deliver content to users. Rails builds on a three-part structure of "models," "views," and "controllers" 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, and controllers deliver data requests to the proper destination so that views can be rendered properly. While there are hundreds of contributors to Rails, there is a smaller core team of established developers who collectively decide on the future of the project. One member of that team, França, in 2012 submitted a relatively brief proposal to add 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 addition was unnecessary and thus worked against the goals of the project. Several developers questioned the purpose of the code: "Why is this a module? I see it being included a single place [. . .] I'd imagine it's not designed to be used anywhere else, so why bother?" (França et al. 2012), Another contributor asked, "Is there an actual problem this code is solving?" (França et al. 2012). Yet another contributor argued, "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 himself did not enter the comment discussion on the merit of his code proposal, leaving his colleagues unclear as to the *intended* purpose of his submission, even if the end result may have differed from that intent. França's contribution thus clashed with the set of practices that had been established and supported by the majority of vocal contributors to Rails, and his colleagues seemed to be in consensus that it was ultimately not a worthwhile addition to the project.

But David H. Hansson—the initial developer of Rails and a "benevolent dictator" over the project with the ability to overrule the decisions of even the core team—ultimately decided to accept França's proposed code. In his support for the proposal, Hansson noted, "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 served the community well. The remaining discussion hinged on the soundness of Hansson's decision, focused primarily on whether Hansson's own preferences for Rails code style may or may not have aligned 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 goals of the larger Rails community he oversees—to result in an acceptance of França's proposal. The interests of the Rails development community *as perceived* by Hansson and França differed from those articulated by other members of the community, but the code change was accepted regardless of those differences. In other words, the persuasive power of França's code was demonstrably stronger *to the right audience* than its apparent functionality, at least to the general Rails community at the time of its initial submission. The "normalization" involved in community acceptance of França's proposal came not from revision relating to its computational efficiency or apparent appropriateness, but from Hansson's top-down endorsement of its integration into Rails code.

Similarly, standardization efforts by some institutional or project-specific authorities have clashed with individual contributors when idiosyncrasies or preferences to write "unprofessional" code (as defined by those in charge of the project) might potentially work against the goals of the establishment. For example, in early 1998, during the hours before the code for Netscape Navigator (now Mozilla Firefox) was publicly released, there was a push by Netscape's lawyers to "sanitize" the program's code and relevant comments so the texts would appear family-friendly rather than potentially vulgar or obscene. As Jamie Zawinski, a former employee for Mosaic Communication Corporation, noted,

> Largely this [sanitation activity] consisted of making sure we had the legal rights to all the code we were releasing, and making sure every file had proper and accurate copyright statements; but they also made us take out all the dirty words. Specifically, "*any text containing vulgar or offensive words or expressions; any text that might be slanderous or libelous to individuals and/or institutions.*" (Zawinski 2004)

For Zawinski and many others who had contributed to the early releases of Netscape Navigator, the act of sanitation—interpreted by lawyers to be a legal necessity—was one of erasure that removed the character of Netscape-related development from the published texts. Yes, there was a tremendous amount of profanity in the comment lines, but for Zawinski and his collaborators, that profanity was an important component of their programming efforts, as well as being an accurate depiction of the frustra-

tion that often accompanied development activities. For his part, Zawinski published, in 2004, a partial archive of commented lines that had been removed from the first public release of Netscape Navigator (see table 3.1).

None of the comments are particularly heinous, and other than the occasional comment targeting a specific individual ("Lou") or group ("marketing"), most of the obscenities are focused on problems with how particular lines or blocks of code *function*, usually in unexpected ways. Given, however, that Netscape was in a position to have its initial public release of code scrutinized heavily by the public, or at least by individuals outside of the corporate and social structure of the Netscape team, a particular presentation of the company and "releasable code" influenced a move toward standardization of otherwise internal discourse to become a publicly palatable set of texts. That is, complaints about Netscape's code in comment lines may have been perceived (whether by the legal team or in anticipation of public audiences) as metonymic critiques of the browser itself rather than, perhaps, as cathartic expressions of temporary anger or disagreements between individual developers. In any case, the comments were intercepted by Netscape's lawyers in order to circumvent any such potential interpretation.

If Netscape had never released the browser's code to the public, would it still potentially be laden with profanities? Are profanities generally offloaded to other discursive channels? It is difficult to say, and the

Table 3.1. Example comments sanitized from Netscape Navigator 4.x (1998)

File Name	Sanitized Comment(s)
`ns/include/xp_mcom.h`	`/* this sucks, should I really include this here or what? *?`
`ns/lib/layout/edtbuf.cpp`	`// LTNOTE: what the fuck. This crashes sometimes??` `// we are fucked! try something different.`
`ns/cmd/macfe/central/profile.cp`	`// * I have no damn idea why this is so convoluted` `// BULLSHIT ALERT: Get out if I can't call GetSharedLibrary.`
`ns/cmd/xfe/src/PersonalToolbar.cpp`	`// crap from marketing`
`ns/cmd/winfe/display.cpp`	`/* check if Lou is a pindick */`
`ns/cmd/winfe/woohoo.cpp`	`return NULL; //The list file did not exist!!!! Bastards!`

black boxes of proprietary code may cause one to wonder whether *most* software to which the public has no access might appear to be reflections of community character, ideology, value, and so on. Some supporters of OSS projects champion the self-policing nature of OSS communities, with "problematic" individuals being ostracized or outright removed from a given project. But such arguments assume that all readers support similar means of evaluating whether or not a given situation should be dealt with. In the case of Netscape, the company's legal team overrode the implicit values of the programming team to show off, rather than hide, the labor—frustrations and all—that went into the development of the browser.

SELF-PROMOTION AND EXPERIMENTATION

Some individuals involve themselves in OSS development projects either to hone or show off their coding skills, putting their contributions before 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 there are many different reasons for participating in such communities. For example, numerous code contributions have undergone review for—and, theoretically, eventual merging into—the central production of the Mozilla Firefox browser over the past twenty years. 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.

Among the most noteworthy contribution reviews for Firefox is 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" (Gozala et al. 2012). In other words, Gozala utilized the collaborative nature of peer review on his code primarily as a way to *improve* his broader 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 (Gozala et al. 2012). One commenter suggested that Gozala could improve the readability of his code by providing more illustrative examples to explain his intent; Gozala responded by suggesting the commenter "just read docs [documenta-

tion], 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. (Gozala 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 the project, offering feedback that could extend to code efforts beyond the scope of the Firefox browser. At no time did the community suggest that Gozala *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 Gozala 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, Gozala'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.

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 (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. A related concept is the programming maxim often referred to as "Linus' Law," named 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, although it has been observed that in practice, many bugs are overlooked or otherwise unidentified (Kerner 2015).

EVALUATION OF CONTRIBUTIONS AND CONTRIBUTORS

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 purportedly meritocratic, and thus beneficial, makeup of their community structures; for example, Kelty (2008) discussed Wikipedia's commitment to meritocracy as a response to systems in which background or professional position can influence one's community standing (345–46). Such efforts can promote a more inclusive and inviting atmosphere for contributors who might otherwise be hesitant or resistant to participating, since in-depth knowledge of code can often appear to be a substantial barrier 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 a 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 [*sic*] 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 claimed to have prospered because it viewed social "power" as a nonprecious "resource," meaning that theoretically any and all volunteers could join in to help as desired. This freedom to participate, however, *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 with social authority 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 brief examples begin to demonstrate how particular individuals and groups may attempt to rhetorically position their work for various audiences and purposes. The goals and their discussions provided above will be compared below with developers' rhetorically aware appeals to determine how frequently such efforts are explicitly employed to achieve a given end versus appeals that implicitly suggest code as the sole focus of deliberation.

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. 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?" (84–85). Ford (2005, 2015) also appeared to recognize implicitly the rhetorical qualities of interfaces and code alike as ways of understanding more effectively the history and activity of software development.

Ford's (2005) final question, which suggested a kind of rhetorical oscillation, is integral to code practices in that it emphasizes the suasive nature of effective code and communication: "What is *sprezzatura* for the Web?" (2005, 91). Sprezzatura refers to one's ability to make a difficult act *appear* as though its execution were effortless, what Castiglione (1988) described as "a certain nonchalance which conceals all artistry and makes whatever one says and does seem uncontrived and effortless" (67). Ford's question effectively highlights concerns of rhetorical style and delivery: How can the effective and successful construction of code—which itself might appear to be not much work—make software development and use easier and more accessible for all? In essence, the rhetorical value of user and coder goals should influence development practices 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)[.] (17)

Despite the increase in speed and machine efficiency gained as a result of the programmer's decision, 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, 17). It is important to note that while the approach provided by Platt is mechanically 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—are ultimately chosen over the diverse interests or needs of the user, which might very well include or assume automatic saving of data to the hard disk.

While developers may not always, or even often, draw attention to suasive strategies that do not focus solely or primarily on code, *when they do*, those strategies can provide rhetoricians with tremendous insight into those developers' aims and how they attempt to induce their colleagues to action. A programmer may not frequently make explicit reference to the language of rhetoric (e.g., almost no one will outright mention *ethos* or *chiasmus* in their comments) but they might mention their credibility (e.g., their experience with a project or the code languages used in it) or personal emotional investment regarding a particular issue. When developers lean on fundamental rhetorical appeals in these ways, the ends to which they work and their perceived relationships with other developers or their projects become visible in intriguing ways for scholars and professional practitioners interested in relevant communication practices. In addition, these rhetorical activities can enlighten us as to the ends to which particular code texts and paradigms are constructed and promoted as well as to how they might be received by various audiences.

Rhetorical Appeals Used in Community Discussions

Developers' use of specific rhetorical appeals and strategies to convince developer audiences to act varies between individuals. There is no *fundamental* difference between using the basic appeals of ethos, logos, and pathos as part of either a code-based or a more discursive effort at constructing meaning, although the types of constraints for each mode of communication may sometimes be distinguished easily from those of the other. Further, it should not be suggested that these fundamental appeals

are the only means by which one might approach code and relevant discourse in order to identify rhetorical activity, but they serve as a productive starting point to engage code-related communication.

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 in a given community as valid support for community-related claims. Developers' *recognition* of their rhetorical efforts is important here, since the ways they construct and present their arguments can be as insightful about their values on coding as is information in the code texts about which they deliberate.

ETHOS

Among the appeals used relatively explicitly in development communities is that of ethos, which is used not only in its classical sense 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, whom they may or may not know well or at all: "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?" (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 the author's character, demands a consideration of ethos focused more on the *performance* of credibility through the presentation of pertinent 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, headius 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 Ruby modified to run inside the Java Virtual Machine, and which thus could work alongside the Java language; 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, but only 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" (headius et al. 2012). Since the project was still "in process," the bugs that JRuby developers might continue to find, when identified by numbers according to their existing maintenance scheme, could make fixing those bugs easier.

headius's offer was 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, who had their own maintenance preferences. 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 et al. 2012). Here, the ethos of the JRuby community clashes with the ethos of the project itself (regarding whether it was worthy of being included in the main Ruby project), bringing to the spotlight the differences between the values of each community.

A counterexample 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 being more relevant 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). The "can of worms" perspective appeared to anticipate other potentially unrelated issues that, should this first problem be considered seriously, the community might then feel obligated to support or address. 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: was it Firefox's domain or that of other JavaScript libraries? No consensus has formed in answer to these questions, in part because no individual has assumed a position of authority to definitively mark out that territory as belonging to jQuery

or another group, or even to "formally" assign that work to another group that may similarly distance itself from the issue. Nonetheless, the community's demonstration that their boundaries exist is significant. Here the developer's ethos can be as much about understanding the *limits* of individual and group authority and expertise as much as understanding when to emphasize the strengths of those qualities.

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 *initial* 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 as support for their 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" (279). For discussions surrounding 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 and initial developer of the software versioning program git. What made this particular request *notable* is that Torvalds adamantly refused to address any code submitted to him 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. (WNeZRoS 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" provided an emotionally powerful articulation of his reasoning. As many others observed, however, the proposed code change was *three lines* long, a trivial amount of code to absorb into the project. The change would in no way require any substantial effort on the part of Torvalds, or of any other developer with the appropriate authority, to accept the code 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 *waste* as a description of how GitHub works (presumably, in contrast to git itself). 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[.] (WNeZRoS et al. 2012)

While Torvalds expounded upon why he felt GitHub failed as a version control system on top of git, his equating the website with "garbage" or "pure crap" remains the most compelling appeal for his position in that it reflected his purpose in rejecting the submission; Torvalds exercised his ability to critique GitHub and its supporters. 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" (WNeZRoS et al. 2012). Further discussion on GitHub's implementation of git's code merge request capabilities revolved around the stylistic preferences for their use held by each participant in the conversation. Few comments were dedicated to logical argument about GitHub's use of git; most of these cen-

tered on the "objectively best" practice that would benefit not only Linux development but all software development hosted by GitHub.

Ultimately, WNeZRoS's proposed code was never implemented through his GitHub submission, but Torvalds' *pathetic* arguments clarified for many developers how continued community work on the Linux kernel would likely 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" (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. But since his complaints directly addressed that system, an inordinate amount of discussion focused on whether or not Torvalds' criticism was rationally 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 and executed, his position was essentially impervious to colleagues' 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 any consensus on the most appropriate means of presenting an argument for a particular community. The kairotic qualities of a coding situation influence each developer and audience in unique ways, and the appeals used by a developer to effect change in his or her audience highlight that developer's understanding of the relationship he or she has with the project under discussion.

Nontraditional Rhetorical Activities Surrounding Code

Online forums and email lists are among the most popular, discursive, and accessible forms of communication among developers in addition to code comments, versioning commit descriptions, and other "code-adjacent" texts. These forms of communication, which both make use of code texts *and* noncode 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 the project's trajectory; in oth-

ers, the goal is to merge 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.

FORKING

The concept of *forking* contributes significantly to the ebb and flow of many development community life cycles (and especially to 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 OSS forks include Ubuntu, a fork of the Debian Linux OS; 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's desiring to tinker with a program outside 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 pessimistic example, Howison (2006) has argued that in most cases, even when projects may have regular forking events occur, there are often efforts by the forking developers to make available improvements for the original program as well as for their own forked versions. This is likely because continued intercommunal goodwill increases the chances of other developers contributing back to the fork in kind.

Despite its technological focus, forking bears a close resemblance to the back-and-forth of conversation. As ideas are circulated 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 nonpreferred 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. (110–11)

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 (social) "testing" of merit, reception, and so on. Contributors to the discussion—individual scholars or citizens—add their voices when they can, but the discussion continues, and often changes, long after each departs from it. The same is true of software development: forking serves as a means of "testing" a set of ideas by allowing motivated individuals to explore alternatives to an original development plan. The strength of a particular argument (i.e., development philosophy) then ultimately is determined by the success of a particular fork as accepted and taken up by a given community. 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 computational strength and efficiency of the program's code base.

The practice of open source software distribution and access could itself be viewed as a kind of rhetorical imitation: At what point does the use of another developer's code blur into the substance of a subsequent developer's work? The use of someone else's efforts 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" (141). *Topoi* are useful to assist a rhetor in con-

structing a particular argument by revealing and generating paths for that rhetor's expression from the potential lines of reasoning he or she could 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 pre-existing form of a given project in order to facilitate inventive activity, 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 originating project or code base. 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. Conversely, a fork may have been created with the intent for it to be separated entirely from its originator as a distinct project in perpetuity. But 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 avenue 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 and orientation are expanded. Even though many forked projects' developers contribute to their originator projects, there is no inherent obligation for them 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.

PUSHING AND PULLING

Many software versioning systems, like git, make use of a two-pronged form of code sharing called *pushing* and *pulling* 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 rhetorical action in a broad sense, 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. (2)

In other words, a successful attempt at suasion occurs when there is a well-navigated, optimally ethical, path balanced between overt manipulation of or pandering to an audience. 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 (ix). For collaborative software projects, this comparison is obviously apt and especially valuable. The means by which a balance between manipulation and pandering is perceived to be attained, however, may vary wildly between communities. The potential interactions that may take place through code-related pushing and pulling, described below, simultaneously draw attention to, and obscure, rhetorical negotiations on inducing developer audiences to accept and improve upon particular code-related decisions and influence the direction of a given development community.

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 employ 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 that 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. (The employment of pushed code as described above is a *kind* of pulling.) Garsten (2006) referred to this sort of approach (in a general sense) as "pandering" in that it demands of the rhetor-developer an orientation that downplays 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 freely redistribute (or fork) 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 is likely to 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 (some of the examples provided earlier in this chapter centered on pull requests and the justifications for their acceptance). This process provides a voice to the individual by giving him or her the opportunity to demonstrate to a community the rhetorical and computational power of a proposed code change. At the same time, if and when the code is merged, whether immediately or following further revision, the identity of its author is partly erased; the code content becomes a part of the main program and, barring any specifically added comments to identify its creator, the code is no longer distinguishable from the other lines of code surrounding it or other files accompanying it. Given the hierarchical structures that constitute many OSS communities, 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 often occurs through successful rhetorical appeal: the user is 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 one or more develop-

ers "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 recurring) model of hierarchical structure that influences subsequent innovation 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, 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 complex rhetorical ecology? It appears 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 as separate and distinct projects, and this ultimately provides an intriguing, and possibly unique, model of rhetorical appeal through the avoidance of direct conversation.

PATCHING

Other than forking and collaborative versioning, perhaps the most "social" practice 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 and attempt to fix code exploits on their own installation of a given program, which are then perhaps distributed out to the broader community of users 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" (85). That is, software patches are generally only applied voluntarily 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 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 prompt users to update relevant programs, either individually or as a "batch."

While this increased automation makes implementing official patches easier, there is an issue of relevant control, not unlike the issue with Microsoft Notepad addressed by Platt earlier in this chapter. In one sense, code concerns become a nonissue 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, or expectation of, users to examine proposed updates on their own and determine whether or not to apply those patches. 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 and its assumed authority in distributing only patches that have passed testing. With manually applied patches, there is an expectation that a user recognizes a particular issue and actively seeks 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 may leave his or her software open to other vulnerabilities (although, in a sense, all software is open to unacknowledged or undiscovered vulnerabilities). Second, the user could specifically avoid 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 increasing its size and processor use with only the additional functions and protections he or she desires. 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. Put another way, the Ship of Theseus will never become fully watertight.

Accordingly, not all patches or updates are equal. In some cases, they are designed to fix seemingly trivial but potentially devastating issues. For example, Valarissa et al. (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 lo-

cate 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 potentially significant consequences for users and their systems, making a "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 that a user will back out of (or "downgrade") the update, making software versioning (the use of differently updated iterations of a program) a particularly intriguing 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 different 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 the software versions they are currently using.

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 or the larger Drupal project 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 pro-

gram 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. Differences in software versions of both drush and git between individual Drupal developers, however, can quickly lead to 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. (2011), the ways git and drush (specifically, a subcommand of drush called "drush make") worked at the time disrupted the successful progress of development for either program, and the ensuing discussion provided an 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 defined the location of a "working" directory in which development would occur, although the **`--bare`** flag specifically changed the hierarchical directory structure of the code project. Functional differences in multiple versions of git and drush used by developers, however, resulted in multiple patch failures. Some users received error messages, while others received successful reports, only to discover the code they *thought* had been patched had not actually 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. But problems persisted: those who reported using git versions 1.7.0.x continued to encounter the issue, 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 might vary wildly in focus and scope toward 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 problems with software, successful and unsuccessful, 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 than those held the community at large. As with any collaborative development activity, rhetorical affordances emerge as being particularly apt (or not) for certain ends and types of action, whether that may be radically altering the trajectory of a given project's development or merely communicating clarification about the intended purpose of some contributed code lines.

Conclusions

Discursive and nondiscursive collaborative activities surrounding code development, from commenting to forking to patching, 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 ef-

fective 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 compel 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 of the web browser Mozilla Firefox, a program collaborated on by thousands of programmers over the past seventeen years. The browser's code, in all its iterations over that span of time, and the development thereof have rhetorically impacted subsequent development, *as well as use*, of the program for particular ends. Where rhetorical action was explored as occurring in relatively conventional text and activities in the examples provided earlier in this chapter, the forms of engagement undertaken through code do not always closely resemble traditional discourse. Nonetheless, they 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

Developing Arguments in Code
The Case of Mozilla Firefox

While the overwhelming majority of critical inquiry into the rhetoric of digital media and technologies has focused on key forms of conventional discourse—those that often surround and describe code practices and texts that make up digital media and technologies—there is also a territory ripe for further investigation: those very code practices and texts. That is, if we are to understand the expressive products of code as rhetorical, we may well find it useful to know how code rhetorically facilitates those expressions. So, in order to clarify a rhetorical understanding of code as valuable object of and site for study, I will embark on an initial, albeit in-depth, analysis of code serving as and creating meaningful communication (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, whose code is composed primarily in two high-level programming languages (C++ and JavaScript)—will serve as the specific object of study. Part of the reason Firefox is an excellent case study is because the code languages used to create it reflect, relatively faithfully, the syntactic and grammatical structure of English, making the example code texts discussed below more accessible than if they were written in a more conceptually abstract and less recognizable or accessible programming language.

As a rhetorical artifact, Firefox is made up of numerous parts: a software program's source code, its compiled "executable" files, the conven-

tional 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* its related commentary. After all, a software program exists more *in code* than in discussions *about* its code. Thus, each of the approaches to meaning making 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 corhetors to initiate, participate in, or otherwise facilitate certain types of action.

As a result, it is necessary to investigate code, as distinct from code-related discourse, 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 meaning constructed in and communicated through 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 the sort of user-centered interactive software engagement that occurs when the program is executed or interpreted by a given machine. The relative accessibility of the languages making up Firefox (C++ and JavaScript) will provide significant assistance in helping the rhetorically minded but potentially code-illiterate reader to recognize how arguments within the code operate toward various suasive ends.

To perform this analysis, I 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 nearly two decades, and thousands of programmers have been involved in its development, writing even more thousands of iterative revisions and additions to the Firefox code base. Thus, the sorts of rhetorical efforts made in the browser's code can tell us a great deal about how members of the development community have effectively communicated with one another about their preferred means of structuring the browser so as to enable particular types of activities.

Further, Mozilla 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. As a result, it is nearly impossible to explore the program's code

comprehensively, although some scholars interested in software, code, and culture have investigated its potential, most notably Black (2015). As Black has noted, "any attempt to critically read source code faces limitations of scale and code when applied to modern application software that is comprised of dozens of modules and millions of lines of code" (n.p.). For a program like Firefox, which has had numerous versions over nearly two decades, these limitations are compounded significantly for critics. In this chapter, I seek not to provide an exhaustive response or realization of others' inquiries but to extend some of their questions, as particularly useful, toward rhetoric-oriented critical analysis.

The Mozilla Firefox 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 allows it to be downloaded and modified by any interested and motivated individual for his or her own purposes. 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 such strategies 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 and discussion about the browser, as well as how those decisions impacted the makeup of the development community invested in continued work on the project.

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 influencing developers' code styles, programming behaviors, and motives for participation. By this I mean that while Mozilla's code is fundamentally rhetorical, many of the examples included in this chapter are unlikely to appear *explicitly* creative or easily identifiable as rhetorical. One can consider the difference between a mundane conversation and a State of the Union address delivered by a US president: both are filled with efforts to construct and communicate meaning, but the latter text is generally described as more rhetorically powerful or more likely to be per-

formed by a skilled orator. Programming as meaningful work, like writing or speaking, is often defined by the strictest and most instrumental 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" (6). Despite this relatively limited view of what code is and what it is for, there have been insightful developments in several related 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 induce scientific or public audiences to action often hinge on tools and strategies that may initially seem *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 compel scientific audiences at particular moments in history to acknowledge the truth of their authors' theories" (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, much of the rhetorical importance of a given code text, as with other forms of text, occurs as much in the sort of action and activity it induces as in the specific content of any individual text. As Muckelbauer (2008) has argued, the very nature of suasion "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" (18). In order to understand code as a type of rhetorical communication, we need to recognize its potential for this kind of 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 comprising 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 arguments in code. Keith (1997) described rhetoric—when considered as an ar-

chitectonic or "design" art—as "concerned with design of language for purposes both individual and social" (234). Where Keith has mentioned *language*, we should extend that thought beyond conventional definitions of language as discourse 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 via the vocabulary of conventional rhetoric, and it may possibly be fruitful for us to embrace only some of rhetoric's terminology to describe and comprehend code as "rhetorical." 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" (75). That said, Gaonkar's argument does not suggest that 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 a program's code, including its logic and how that logic is constructed so as to be expressed effectively to various human (and potentially nonhuman) audiences?
2. What sorts of appeals are offered in intracode communication, specifically in noncomputational comment lines of text "explaining" code functions above, below, or otherwise near 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 replicable across multiple situations in ways that more empirical methodologies might facilitate. Inquiries into these concerns can nonetheless offer critical understanding into the complexities of suasion 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 a broader comprehension of, and ability to employ more effectively, rhetorical communication and the strategies available to a rhetor.

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 Safari), with an estimated 9.3 percent share of the browser "market" as of December 2017 (W3Counter 2017). Firefox is also arguably the browser with the greatest amount of development taking 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 as freely available. For example, source code for the Chrome web browser is not publicly available, although source code for the OSS browser project Chromium, on which Chrome is based, *is* publicly available, albeit with some different features and licensing parameters than those of Chrome. Because Firefox exists as an open source project, nearly all its code and discourse surrounding its development is publicly accessible, which allows interested parties to explore the trajectories of development, collaboration, and conflict that have occurred over a decade of activity. As a result, Firefox serves as a rich site for critical 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.

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 Mosaic, one of the very first Internet browsers. In fact, the name "Mozilla" 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, and Walton 1999). While Navigator was offered to the public at no charge, it was not initially popular 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, in January 1998 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 nonprofit 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 became 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 that release adhered to the original license, that is, its source code *also* made freely available alongside the compiled, executable version of the browser. Communication (i.e., email) related to the program's development also became publicly available, meaning that any interested parties could browse or engage in discussion about the project and involve themselves in code *and any other* communicative efforts to influence Mozilla's developmental trajectory. The semiorganized, semichaotic nature of the developer community for the Firefox browser, as demonstrated through members' writing, 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 similarly connected but distinct 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. But with almost seventeen 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 earlier, although the voluntary and open nature of the project 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, such as Black (2015), who performed topic modeling analysis to explore various sociocultural influences on development of the browser. 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 communicated and what it induces, inquiry into the culture of development might involve exploring (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 occasionally 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.

The Turn to Open Source Software: Ramifications of Firefox's Development

By transforming its software development process from a corporate to a community-based project in the early 2000s, Mozilla had radically redefined its general 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; instead, the Mozilla Project's new organizational scheme meant that the product *and* the process were open and available, not just for consumption but for further development. This is not to suggest, however, that Firefox's development is fully egalitarian, democratic, or even meritocratic, even though the Mozilla website identifies the project as the last of these (Mozilla n.d.a). The project consists of 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.

Despite claims of meritocratic structures and mechanisms, the existence of a hierarchical administration in regards to Mozilla's software programs has the potential to work against the values of an otherwise "open" approach to community-based collaboration efforts. As Hamerly, Paquin, and 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 that make up 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—and 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 impact(s) 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 structured or authoritative one, it is important to recognize how these decisions constrain the development process.

SOFTWARE VERSIONING SYSTEMS AND RHETORICAL ACTION

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 that emphasizes the specific line-by-line differences that exist in each iteration of the code files. Each developer maintains his or her own 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 a developer might feel are worthy of inclusion in the official version of the program are submitted 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. As of September 1, 2016, Mozilla currently has almost 150 developers listed publicly as members of its organization, participating in more than three hundred distinct projects, and using two major software versioning utilities to track browser development.

Each of these utilities is fueled by a distinct exigence, although the individuals contributing to either are spurred by their own motives and sense of kairos. **Mercurial** 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**, meanwhile, 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," sending 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," submissions for incorporation in 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 (fewer than 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, which is built on git, a different version control program (Mozilla 2012b) argued

by some to be more flexible; Thomson (2008) referred to git as MacGyver and Mercurial as James Bond, with the former capable of doing almost anything but the latter being exceptionally skilled for the "right" tasks. The GitHub website provides a system for easy tracking, commenting, and forking (making new, individually customizable versions) of git-based projects. For its "Gecko-Dev" GitHub repository, which includes a significant majority of the code for the Firefox browser, Mozilla has over 1,000 forks, potentially distinct branches from the official repository "tree," each maintained and updated according to the individual schedules of the developer(s) who initiated a given fork. Many of those forks serve to incorporate, hierarchically, changes made to other projects, through git or Mercurial, that affect the Gecko-Dev 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. One of GitHub's primary appeals, beyond its relative ease of use, is the number of users and repositories it hosts. GitHub staff reported that by the end of 2012, more than two million users were collaborating on more than three and a half million projects (GitHub 2012). It is thus safe to say 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.

Mercurial and git, however, are not the only tools used for Firefox development. Mozilla's Treeherder utility provides developers with information about the results of their individual automated browser builds and the tests run on those builds (Mozilla 2016). Essentially, Treeherder functions as a collaborating corhetor, working alongside human developers to evaluate the functionality and performance of individual builds (and thus the individual code contributions leading to those completed builds). Treeherder's agency lies in its "gatekeeper" role, as it checks and potentially rejects code proposals that do not pass the tests it runs on developers' browser builds. If a proposed code change does not comply with the goals of the Treeherder tests, it is likely to be rejected by the human administrators of the project, even if the goals of that code are valued by them (although subsequent revision of a proposal may ultimately result in its acceptance). The code running Treeherder has a powerful rhetorical impact on the code for Firefox, arising out of the project's development practices, through human developers modifying the latter, based on meaning (i.e., 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), Treeherder is even more direct in its activities: it provides automated review *itself*, implicitly passing judgment on the potential value of a code change. That is, the utility becomes an active agent participating in the rhetorical activities of software development related to the Firefox browser. The vast majority of the review "judgments" reported are mechanical failures, code unable to compile due to some error in one or more operations within the compilation process. What makes the judgments noteworthy is that regardless of whether or not a proposed contribution works on a developer's machine, the output error messages provided as a result of Treeherder review suggest to the developer that his or her code is faulty or otherwise unworthy of inclusion in the set(s) of proposals to be discussed within and scrutinized by the more substantial Firefox development community. For example, Rahm et al. (2016) identified, discussed, attempted to replicate, and then subsequently developed a variety of potential solutions to a bug related to playing audio in the browser. Multiple contributors confirmed that the problem existed, clarified among themselves what, exactly, the problem seemed to do and what it affected, and then tested the assorted proposed solutions that various individuals offered. As Treeherder reported successes or failures with the proposals, contributors adjusted their solutions accordingly. Eventually—just over two months after the initial bug report filed by Rahm—one solution was deemed to be effective and worthy of incorporation into a browser update. Thus the Treeherder program, in other words, serves as a kind of prototypically situated machine-involved audience to whose specifications all Firefox code must successfully adhere. At the same time, Treeherder is an active rhetor that promotes a specific form of action: suggestions for improvements to be undertaken by developers so as to improve the Firefox browser.

To an extent, a recognition of Treeherder as an important audience collapses much of the rhetorical potential regarding Firefox's collaborative development into a standardized, "sanitized" process in which the default product—the browser download that is emphasized on the program's website—is valued far more than any of the customized, nuanced attempts to experiment with the software, attempts in which some 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 Treeherder is one in which experimental play or discovery is

not esteemed as highly as focused development toward 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.

Rhetorical 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. The social quality of collaborative development, however, is not limited to computational or efficiency-based optimization of a given program's code, or even of code in general. 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 continually fueled by a variety of 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 induce one another to act, in different ways and to different ends. Miller (1984) observed that genre serves as a means of rhetorical action constituting and reconstituting a discourse community through its recurrent 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, 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" (97), we can see numerous purposes and efforts at work, sometimes harmoniously and sometimes in dissonant and competing ways. Spinuzzi (2003) described an even more complex framework of the *genre ecology* as

"a descriptive model of compound mediation [. . .] highlight[ing] idiosyncratic, divergent understandings and uses of artifacts and the practices that surround them as they develop within a given cultural-historical milieu" (n.p.). The genre ecology as a concept is especially valuable for understanding software development as a rhetorical activity, thanks to ecological recognition of and attention to generic artifacts as diverse as code libraries and scripts to user manuals and face-to-face conversations between developers.

For developers, the various 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 and reify their contemporary professional and community-based identities. Individual iterations of these genres simultaneously contribute to the confirmation of particular genres while also reconstructing them according to ever-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 rhetorical, and not merely formal, components of these genres. Multiple developers in an OSS community work together *in specific ways* to create a program: a 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? As an initial response to this question, Murray established a five-point "non-discursive theory of writing" to acknowledge a set of values that are nondiscursive but nonetheless potentially significant for communicating discursively as well: "will-to-image," "will-to-improvise," "will-to-intuit," "will-to-

juxtapose," and "will-to-integrate" (140–41). Murray's question, and the kernel of his nondiscursive writing theory, are key for any sort of recognition of the rhetorical genres in which developers regularly work, especially those outside the bounds of what is conventionally recognized as discursive communication. For Murray, 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 allowed rhetors to disregard explicit reasoning in favor of intuitive invention and play so that they can "generate connections as they compose multimodal texts," whether as students in the classroom or otherwise (188). For code (in comparison to images), the symbolic logic of computation itself can be acknowledged as a nondiscursive system of symbolic meaning and action; as with image and many other forms of communication, when it comes to understanding expressions within a system, there is often a barrier to "entry" for individuals unfamiliar with that systems or the forms of communication used within it. Further, meaningful activity is not just possible but *inherently present* in any effort to communicate through the system.

By building on this recognition, 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. Brown (2015), in looking at software exploits and building on the work of Gaonkar (2004) and Streuver (2009), has argued for a definition of rhetoric as an art interested in what is most widely "possible" in a given situation, in contrast to many traditional interpretations of rhetoric as focused on the "probable" (Brown 2015, 75–81). "Potential," then, might best be understood as an oscillation between possibility and probability, the moment when a rhetor—or any of the influential factors involved in a particular rhetorical event—considers possible means and decides which of those to pursue, based on probable success of inducing particular ranges of action (with "success" interpreted in any number of ways). 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 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.

The two genres discussed below certainly recur across numerous code projects, iterations, and authors, but they are not always *typified* in easily recognizable ways; that is, the formal or structural components of many genres do not lend themselves easily to code texts written in different languages or for different communities whose style preferences may radically vary from those of other communities (cf. an examination of code genres via Drupal modules by Brock and Mehlenbacher 2017). The *purposes* for particular recurrent responses to given situations are certainly recognizable, as are the kinds of action that rhetors may attempt to induce through their communicative efforts. As a result, these genres may best be understood as responses to tensions surrounding recurrence in Firefox, namely, as individuals and groups attempt to entrench or continue particular traditions and normalized behaviors while also, or in response, presenting innovative or unconventional approaches to solving particular problems or answering particular questions considered significant to the development community.

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 proposals are trivial in nature, extending functionalities of the browser in specific and situational ways, sometimes focused more heavily on code practices and style (e.g., altering variable or function names or logic to suit readability within the broader development community), while at other times focused on the expressive possibilities of the program's code (e.g., how the user experience will change, and to what ends it will change, when a particular adjustment to the program is made). In any case, proposed contributions to the Firefox code base are often themselves further developed and revised in order for the community to determine just how beneficial, if at all, such a set of contributions might be to the project.

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' rhetorical influence upon other developers as the clearly identified audience. Despite this focus, however, the proposals generally possess qualities that reflect practices related to large-scale development activities. For example, we can see in table 4.1 a relatively recent adjustment to a development script the inno-

vation upon an existing feature, as part of a tool called the Device Manager Android Debug Bridge, or ADB, 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. Originally, the ADB code (written in the Python scripting language) enabled a developer to upload directories of files from the local hard drive to an external server. As noted by gbrownmozilla (2011) in the comments at the beginning of the quoted excerpt (lines 95–97), however, the code as originally written did not account for symbolic links, links that pointed to other directories and that may not have been intended to get included in the set of data to be uploaded or that may not even exist (in an accessible location or even at all) on the remote server. gbrownmozilla modified the ADB code to work

Table 4.1. Proposed Android ADB change by gbrownmozilla (2011)

Line #	Code by gbrownmozilla (2011)
94	`def pushDir(self, localDir, remoteDir):`
95	`    # adb "push" accepts a directory as an argument, but if the directory`
96	`    # contains symbolic links, the links are pushed, rather than the linked`
97	`    # files; we push file-by-file to get around this limitation`
98	`    try:`
99	`      for root, dirs, files in os.walk(localDir):`
100	`        relRoot = os.path.relpath(root, localDir)`
101	`        for file in files:`
102	`          localFile = os.path.join(root, file)`
103	`          remoteFile = remoteDir + "/"`
104	`          if (relRoot!=". "):`
105	`            remoteFile = remoteFile + relRoot + "/"`
106	`          remoteFile = remoteFile + file`
107	`          self.pushFile(localFile, remoteFile)`
108	`        for dir in dirs:`
109	`          targetDir = remoteDir + "/"`
110	`          if (relRoot!="."):`
111	`            targetDir = targetDir + relRoot + "/"`
112	`          targetDir = targetDir + dir`
113	`          if (not self.dirExists(targetDir)):`
114	`            self.mkDir(targetDir)`
115	`      self.checkCmdAs(["shell", "chmod", "777", remoteDir])`
116	`      return True`
117	`    except:`
118	`      print "pushing " + localDir + " to " + remoteDir + " failed"`
119	`      return False`

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, gbrownmozilla 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.

gbrownmozilla's innovation reflects broader practices in that they made use of multiple loops in order to ensure that the exact *sort* of outcome he anticipated was likely to occur. Before gbrownmozilla's proposed code change, there was no check to determine whether or not all files being uploaded were "valid" and not security risks. gbrownmozilla's ordering of relevant operations, complemented by informative commentary, induces other colleagues to consider the implications of the ways 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, gbrownmozilla has distinguished between them, but because they have similar qualities, those tests are repeated in order to lessen the chance of an error occurring. In other words, gbrownmozilla has positioned readability and clarity in intent over optimized computation so that the tool they worked on is not only improved by that contribution but that other developers can see how and why that contribution has a positive impact on the browser's development. It is certainly possible that some of gbrownmozilla's colleagues will interpret this code as inelegant or inefficient, but its value in communicating what it does clearly and accessibly, so that it might be improved further by other contributors, is arguably greater than any lost machine efficiency in its current form.

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 relevant browser code allowed for one of two events to 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, Goodger, and Romano 2006a). This functionality, however, was improved multiple times. One early improvement was 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, Goodger, and Romano 2006b). A much more recent innovation made use of the browser's since-expanded capacity to provide an initial home page via the browser's about protocol, written almost immediately above the untouched 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 4.2.

Table 4.2. Three iterations of Firefox startup home page code (2006a, 2006b, 2010)

Line #	Code by Walden, Goodger, and Romano (2006a)
69	`var useCurrent = document.getElementById("useCurrent");`
70	`var chooseBookmark = document.getElementById("chooseBookmark");`
71	`var bookmarkName = document.getElementById("bookmarkName");`
72	`var otherURL = document.getElementById("otherURL");`
	`[...]`
80	`if (bookmarkName.getAttribute("uri") == " (none)") {`
81	`  useCurrent.disabled = otherURL.disabled = true;`
82	`  bookmarkName.disabled = chooseBookmark.disabled = false;`
83	
84	`  return "bookmark";`
85	`}`
86	
87	`var homePage = document.getElementById("browser.startup.homepage");`
88	`if (homePage.value == homePage.defaultValue) {`
89	`  useCurrent.disabled = otherURL.disabled = true;`
90	`  bookmarkName.disabled = chooseBookmark.disabled = true;`
91	`  return "default";`
92	`}`
93	`else {`
94	`  var bookmarkTitle = null;`
95	
96	`  if (homePage.value.indexOf("|") >= 0) {`
97	`// multiple tabs—XXX dangerous "|" character!`
98	`// don't bother treating this as a bookmark, because the level of`
99	`// discernment needed to determine that these actually represent a`
100	`// folder is probably more trouble than it's worth`
101	`  } else {`
102	`#ifdef MOZ_PLACES`
	`[...]`
115	`#else`
	`[...]`
124	`if (bookmarkName.getAttribute("uri") == homePage.value)`
125	`  bookmarkTitle = bookmarkName.value;`
126	`#endif`
127	`  }`

Table 4.2.—*Continued*

Line #	Code by Walden, Goodger, and Romano (2006b)
72	`var win;`
73	`if (document.documentElement.instantApply) {`
74	`  // If we're in instant-apply mode, use the most recent browser window`
75	`  var wm = Components.classes["@mozilla.org/appshell/window mediator;1"]`
76	`                        .getService(Components.interfaces.nsI WindowMediator);`
77	`  win = wm.getMostRecentWindow("navigator:browser");`
78	`}`
79	`else`
80	`  win = window.opener;`
81	
82	`if (win) {`
83	`  var homePage = document.getElementById("browser.startup. homepage");`
84	`  var browser = win.document.getElementById("content");`
85	
86	`  var newVal = browser.browsers[0].currentURI.spec;`
87	`  if (browser.browsers.length > 1) {`
88	`    // XXX using dangerous "|" joiner!`
89	`    for (var i = 1; i < browser.browsers.length; i++)`
90	`      newVal += "|" + browser.browsers[i].currentURI.spec;`
91	`  }`
92	
93	`  homePage.value = newVal;`
94	`}`

Line #	Code by Sharp et al. (2010)
77	`syncFromHomePref: function ()`
78	`  {`
79	`    let homePref = document.getElementById("browser.startup. homepage");`
80	
81	`    // If the pref is set to about:home, set the value to "" to show the`
82	`    // placeholder text (about:home title).`
83	`    if (homePref.value.toLowerCase() == "about:home")`
84	`      return "";`
85	
86	`    // If the pref is actually "", show about:blank. The actual home page`
87	`    // loading code treats them the same, and we don't want the placeholder text`
88	`    // to be shown.`
89	`    if (homePref.value == "")`
90	`      return "about:blank";`

```
91
92        // Otherwise, show the actual pref value.
93        return undefined;
94      },
95
96      syncToHomePref: function (value)
97      {
98        // If the value is "", use about:home.
99        if (value == "")
100         return "about:home";
101
102       // Otherwise, use the actual textbox value.
103       return undefined;
104     }
```

The basic functionality of the code—generating a window of content for the user when the browser is initialized—remains relatively stable across each text iteration, but the innovations incorporated by each set of developers, when viewed together, offer a valuable collection of 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 may initially seem to have unrealized potential.

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 (with the implicit assumption that the site was likely to be visited frequently). 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 to the code and its functionality (the update by Walden, Goodger, and Romano 2006b) involved expanding its use—fitting into a space commented on originally but not actually composed in code—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 those demonstrated in the "FizzBuzz" examples in chapter 2. In the case of the home page, the loop iterates 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 (by Sharp et al. 2010) 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 these 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 in style, its generic purpose, if not its form or structure, 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.

NORMALIZATIONS OF CODE PRACTICES

Just as new features are constantly introduced by developers of varying skill or familiarity with a program and tested by the broader community, so too are the code texts constituting those features scrutinized and recomposed so that they adhere to the (admittedly evolving) programming style prescribed by the community. Essentially, the generic conventions being pushed against as part of efforts to innovate are also at least temporarily reified 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 pop-up window is rendered and given focus. Lamouri et al. (2010) provided a JavaScript module for the browser that would cause the program to create, render, and focus on the pop-up window. Two years later, the *same feature* has been reworked, not to alter the feature itself but to reframe its mechanisms stylistically in such a way as to more closely reflect the procedural logic and rhetoric of Firefox development in a broader sense. See table 4.3.

The code is not *drastically* different between these versions, despite dozens of revisions and improvements to the code made between 2010 and 2012 within the file in which these lines appear. The way each of the operations has been restructured is telling: the functionality added by

Table 4.3. Firefox pop-up removal code in JavaScript, 2010 (*upper*) and 2013 (*lower*)

Line # Code by Lamouri et al. (2010)

```
838   // If the user type something or blur the element, we want to
      remove the popup.
839   // We could check for clicks but a click is already removing
      the popup.
840   let eventHandler = function(e) {
841     gFormSubmitObserver.panel.hidePopup();
842   };
843   element.addEventListener("input", eventHandler, false);
844   element.addEventListener("blur", eventHandler, false);
845
846   // One event to bring them all and in the darkness bind them
      all.
847   this.panel.addEventListener("popuphiding", function(aEvent) {
848     aEvent.target.removeEventListener("popuphiding", arguments.
      callee, false);
849     element.removeEventListener("input", eventHandler, false);
850     element.removeEventListener("blur", eventHandler, false);
851   }, false);
852
853   this.panel.hidden = false;
854   this.panel.openPopup(element, "after_start", 0, 0);
```

Line # Code by ehsan et al. (2013)

```
671   // If the user interacts with the element and makes it valid or
      leaves it,
672   // we want to remove the popup.
673   // We could check for clicks but a click is already removing
      the popup.
674   function blurHandler() {
675     gFormSubmitObserver.panel.hidePopup();
676   };
677   function inputHandler(e) {
678     if (e.originalTarget.validity.valid) {
679       gFormSubmitObserver.panel.hidePopup();
680     } else {
681       // If the element is now invalid for a new reason, we
      should update the
682       // error message.
683       if (gFormSubmitObserver.panel.firstChild.textContent !=
684           e.originalTarget.validationMessage) {
685         gFormSubmitObserver.panel.firstChild.textContent =
686           e.originalTarget.validationMessage;
687       }
688     }
689   };
690   element.addEventListener("input", inputHandler, false);
691   element.addEventListener("blur", blurHandler, false);
692
```

Table 4.3.—*Continued*

```
693   // One event to bring them all and in the darkness bind them.
694   this.panel.addEventListener("popuphiding", function
      onPopupHiding(aEvent) {
695     aEvent.target.removeEventListener("popuphiding",
      onPopupHiding, false);
696     element.removeEventListener("input", inputHandler, false);
697     element.removeEventListener("blur", blurHandler, false);
698   }, false);
699
700   this.panel.hidden = false;
```

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 alongside numerous other changes in Firefox's code and accepted coding style(s) during this time period. 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 continues to evolve.

So what exactly is happening in the code described in table 4.3? The procedure is not extremely complicated: when someone closes a pop-up or completes a form in it, the pop-up 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 2013 version of the code from the 2010 version is how individual events have been anticipated, most notably the shift from a general **`eventHandler`** into separate **`blurHandler`** and **`inputHandler`** functions, even while **`addEventListener`** and **`removeEventListener`** remain descriptive of general, catch-all functions. **`inputHandler`** 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 pop-up 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) {`** (ehsan et al. 2013). These possibilities are addressed obliquely in the initial set of comments in the 2013 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.
```

This extra-code explanation offers context as to reason for changes being made to the original version of the pop-up's closing functionality, but it also offers a judgment on that initial iteration: according to the original, the pop-up was removed when "the user type [*sic*] something or blur [*sic*] the element" (Lamouri et al. 2010). While the distinction appears minor, the user's "interaction with the element *and* mak[ing] it valid" (ehsan et al. 2013, 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 "non-industrial," if not incredibly common, 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 2013 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 2013 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. This 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 Firefox community's efforts to norm its practices remain varied and inconsistent in execution, suggesting that the community is further made up of numerous smaller development groups whose members have their own motives and motivations and who may not be interested in working on all aspects of the browser equally.

When a developer is 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 make up 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, although they may not always agree. These efforts are not limited to larger-scale concerns, as many developers offer only minor changes to components of the browser's code, and these changes may resemble flurries of small-scale revisions and subsequent discussion thereof. As a result, that code, when studied across multiple textual iterations, is ripe for examination of how specific rhetorical devices incorporated into code can influence development in particular ways.

Rhetorical Devices in Code

Just as developers are frequently 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 sense, through the code they write. As demonstrated in chapter 2 by the variety of means by which a developer could create a quine, and the delicacy involved in ensuring that it can be interpreted or executed as intended, 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 persuade his or her audience *explicitly* through these code decisions alone. Instead, the potential intent behind his or her code contributions can and should be read as a combination of rhetorical and computational decisions; it is not enough that code might function mechanically but it also needs to be understandable by, and *meaningful* to, an audience who needs to build upon that code to continue developing a program toward particular ends.

REPETITION AND ARRANGEMENT

Those code functions and variables providing critical operations (i.e., those valuable to a number of important tasks) might be referred to repeatedly at crucial points in a larger body of code. Similarly, such critical code may be arranged within particular code files or directory structures

in such a way as to call attention to their significance. Together, developers' use of repetition and arrangement in meaningfully important ways allows them to make potentially powerful rhetorical claims through their code that resemble classical devices meant to describe conventional approaches to repeating and structuring important information. Among those commonly appearing in code texts include *anaphora*, the repetition of words or phrases at the beginning of statements; *epistrophe*, the repetition of final words or phrases for rhetorical effect; *symploce*, the combination of anaphora and epistrophe; and *exergasia*, the repetition of ideas (albeit often in different wording or delivery across each iteration thereof). By repeating a particular point or statement in the same syntactic position or in conceptual relation to another given point, 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.

The notion of rhetorical repetition is also closely related to the idea of *climax*, in which an argument's structure is based on the increasing significance of included statements, with more important ideas and points following the introduction and explanation of less important ideas. Climax is very often 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) anticipated and desired (or, in some cases, unanticipated and *not* desired) by a developer. As a result, the rhetorical activities engaged in by software programmers often attend to the ways of reading code that suggest importance given to procedures based on their apparent repetitive qualities.

Repetition is an incredibly important consideration for any act of communication, as a rhetor can orient an audience toward certain concepts or arguments in particular ways through the skillful (or unskilled) application of repetition. Repetition is also significant for programming for multiple reasons: repeated blocks of code can contribute to inefficient computation through file bloat; inconsistent use of code blocks or logical structures (as different versions of similar code may be called at varying points in a program's code); and inaccurate reading practices as collaborators are forced to decipher what precisely should be culled, revised, or otherwise altered throughout the extent of a program's code texts. In some programming circles, such as the general Ruby development com-

munity, the desire to avoid these issues is articulated through the "DRY" principle or philosophy, DRY standing for "Don't Repeat Yourself" (Matsumoto 2007, 479). In essence, DRY programmers seek conciseness as a means to elegance, although this conciseness does not mean *never* repeating code but rather avoiding unnecessary repetition (e.g., the difference between redefining an object class vs. initializing a single object out of that class). With a number of the world's most popular programming languages making use of modularity such as that of object orientation, the impact that unintentional repetition—or, more accurately, unintentional iterative difference across multiple versions of a given code block—might have on a program's successful execution is potentially enormous.

Rhetorical attention to repetition and arrangement is especially important for a large-scale OSS program like Firefox, since development on the browser involves work on dozens of interlinked modules each of whose code needs to set itself apart from the others. Each of these same modules, however, must also maintain a stylistic and logical 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 fully scalable "fractal" state (wherein the same general structures are perfectly or infinitely repeated at different scales of code), there are nonetheless observable efforts by developers to convince 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). dmose%netscape.com and peterv (2002) 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 existed multiple **`processNext()`** functions, each working similarly but with a different focus: one to move between folders, one to 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 given in table 4.4.

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 subfunction of the **`markFolder()`** function. Here it is initially not *run* but its operational structure is established so that it can be run when called elsewhere in the code (specifically, in line 197 at the end of the quoted excerpt). 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 is called in 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

Table 4.4. Example of Firefox spam filter code in a **`processNext()`** function

Line #	Code by dmose%netscape.com & peterv (2002)
179	`function markFolder(aSpam)`
180	`{`
181	`  function processNext()`
182	`  {`
183	`    if (messages.hasMoreElements()) {`
184	`    // Pffft, jumping through hoops here.`
185	`      var message = messages.getNext().QueryInterface(nsIMsgDBHdr);`
186	`      mark(message, aSpam, processNext);`
187	`    }`
188	`    else {`
189	`      gJunkmailComponent.mBatchUpdate = false;`
190	`    }`
191	`  }`
192	
193	`  getJunkmailComponent();`
194	`  var folder = GetFirstSelectedMsgFolder();`
195	`  var messages = folder.getMessages(msgWindow);`
196	`  gJunkmailComponent.mBatchUpdate = true;`
197	`  processNext();`
198	`}`

the procedural structure of its algorithms, as well as the readability of the above **`markFolder()`** function, 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. Much of this structure arguably stems 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 construction of the **`Word`** class and its related objects in the **`HashMapClass`** example 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 **`process Next()`** 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 occur for creative or professional purposes, stressing a paradigm of practice whose impact extends beyond the specific instance under

scrutiny at any given moment. Such practices, however, are not limited to the placement of significant operations and procedures (in climactic, anaphoric, or epistrophic senses) but are extendable to 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 reflect an awareness of exergasia, which—while related to the devices of repetition and arrangement discussed so far—has the potential to be exponentially more powerful in communicating significance through iteration (and implicit or explicit impact of the variations across relevant iterations).

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 multiple separate functions or sets of operations. Rhetorically, this can be considered a type of *exergasia*, a particular type of repetition: 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 readers on how other developers have determined code structures within the program 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 regards to its facilitated activities.

For a massively collaborative OSS program like Firefox, exergasia is a powerful tool by which myriad developers signal to one another how particular procedures should be constructed and executed across functions, modules, and iterative program releases. By making use of explicit repetition 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 and use of code as rhetorically powerful communication. Given the ability of code to loop iteratively through a set of operations, it is possible, if not inevitable, for repetition to occur *conceptually* but not *explicitly* in the statements that constitute 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" (or arguably "inefficient") repetition is implemented as well as what its implementation can suggest to us about development practices surrounding a program.

Object-oriented programming (OOP) languages have the potential to make especially interesting use of exergasia through the instantiation of unique *objects*. An object, in an OOP language, is a set of data potentially containing any number of variables, functions, structures, etc. and which is built on a larger *class* defining the essential parameters of each individual object instantiated from that class. That is, an object is a bundle of data with properties from a larger class; these properties are inherited from the class and shared across multiple objects. The *use* of those properties by any individual object, however, is distinct from other objects' use thereof. This concept, generally speaking, has its root in Plato's theory of forms: the ideal version of a thing is reflected, always imperfectly, in realized 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 different individuals may perform the same activity in very different ways.

The idea of a human as a helpful example "object" for the purposes of object-oriented programming is so pervasive among developers 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 saskatchewancatch et al. (2016) describe the ways in which object-oriented languages like JavaScript enable practices of iteration and repetition alike through the *potential* constructed through object classes (or, in the case of JavaScript, prototype constructors), using "person" as the most accessible example by which to model object-oriented programming:

In the example in table 4.5, a prototype has been established (**`Person`**) and two specific person objects (**`person1`** and **`person2`**) have been defined based upon its properties. Any adjustments made to the prototype

will affect the **`person1`** and **`person2`** objects, 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 from the same prototype. In this sense, all relevant code is simultaneously repeated across all prototypical instances *and* expressed in potentially unique ways, as each iteration of the prototype differs from most, if not all, of the others. The argument, provided through the use of exergasic code written as 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, saskatchewancatch et al. (2016) 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. (It would be incorrect, however, to fully map or analogize programming objects, prototypes, or classes to humans or sets of human behaviors, characteristics, cultures, and so on. It is a useful metaphor for a brief engagement with the basics of object behavior and inheritance.) 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

Table 4.5. Example of object creation in JavaScript

Line #	Code by saskatchewancatch et al. (2016)
1	`var Person = function (firstName) {`
2	`  this.firstName = firstName;`
3	`};`
4	
5	`Person.prototype.sayHello = function() {`
6	`  console.log("Hello, I'm " + this.firstName);`
7	`};`
8	
9	`var person1 = new Person("Alice");`
10	`var person2 = new Person("Bob");`
11	`var helloFunction = person1.sayHello;`
12	
13	`// logs "Hello, I'm Alice"`
14	`person1.sayHello();`

and socially, not unlike the sorts of repetitive rules outlined in the **`Person`** prototype example.

For an example of exergasia inside Firefox's production code base, troy%netscape.com (1999) 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. But 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`**, customized functions are necessary to make and store those calculations, and troy%netscape.com composed four such relevant function blocks. Because these blocks worked *almost* identically, troy%netscape.com provided a uniform structure across all four of the methods needed to compute their values (i.e., one method each for minimum width, minimum height, maximum width, and maximum height). One of the code blocks appears in his initial text as seen in table 4.6.

This block is made up 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 (in this case) the parent-level component in which the current 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"

Table 4.6. Example of repetition in Firefox's code related to browser size calculations

Line #	Excerpted code by troy%netscape.com (1999), from /layout/generic/nsHTMLReflowState.cpp
1395	`nsStyleUnit minWidthUnit = mStylePosition->mMinWidth.GetUnit();`
1396	`if (eStyleUnit_Inherit == minWidthUnit) {`
1397	`  mComputedMinWidth = aContainingBlockRS->mComputedMinWidth;`
1398	`else {`
1399	`  ComputeHorizontalValue(aContainingBlockWidth, minWidthUnit,`
1400	`  mStylePosition->mMinWidth, mComputed MinWidth);`
1401	`}`

rather than "horizontal" values, and maximum-related blocks look for "max" rather than "min" values.

Because the four blocks are so close in appearance as well as in proximity to one another, there is a relatively clear suggestion that they have related functional purposes. 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 static to dynamically calculated values assigned to variables used throughout the program. 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.

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 rhetorical efforts 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, induce one another in implicit and explicit ways to accept the practices they suggest through the texts they produce.

The act of reading code as a rhetorical text and practice, however, can hold considerable difficulty for the vast majority of rhetorical critics unfamiliar with programming or the languages thereof. Examinations of code like those provided in this chapter may offer helpful examples for future investigations and support for (as much as is possible) more comprehensive and in-depth scrutinies of individual code projects, trends, and genres across multiple programs, or even of programming languages themselves. The possibilities for analysis and application of the rhetorical activities found in code (whether for industrial, civic, or pedagogical purposes) are not necessarily *endless* but are myriad and diverse, and it is in our interests as scholars and teachers of rhetoric to attend to the overwhelming amount

of code being written each day and the means by which its authors communicate meaning to collaborators and other readers through their code.

Mozilla Firefox provides a valuable locus for an in-depth analysis of rhetorical efforts made in code by a massive programming community of thousands over a decade 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. But 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 suasion whose components also shift and develop over time.

In the next chapter, I turn from my focus on analysis to inventive experimentation, offering a series of exercises related to the rhetorical composition of code. These exercises are intended to be accessible to the nonprogrammer audience, and they can be practiced in the web browser rather than requiring any other special software to be installed. Together, these exercises function as a set of *progymnasmata* that introduce fundamental concepts of programming and suggest means of employing those concepts for effective rhetorical communication through the composition of code texts. While the progymnasmata are hardly comprehensive in coverage of rhetorical principles applied in code, they may be valuable for readers who remain skeptical or confused about the actual activity of composing via procedure rather than conventional discourse. Further, for readers with more familiarity with programming, the progymnasmata may be useful exercises for explicitly engaging with particular rhetorical principles during an activity generally considered to be a purely instrumental form of writing.

CHAPTER 5

Composing in Code
A Brief Engagement with JavaScript

While we can engage code from "without" in the form of a conventional scholarly investigation, there remains a missing component integral to understanding how arguments are (and can be) made in code: the activity of *actually composing* code texts. This is not to suggest that one must be an *expert* coder or programmer to develop arguments in code, but relevant knowledge—of procedure, specific language syntax, and so on—certainly facilitates the development of more complex or nuanced communication. Put another way, echoing Haefner's (1999) call for composition instructors to discuss code contexts, functional possibilities, and the relationship between Structured Programming and the hierarchical structures of many American corporations, the practice of composing in code will absolutely benefit our subsequent rhetorical analysis and critique of code if we are to effect change in a world in which it is more or less impossible to escape the influence of software and digital technology.

In this chapter, we will approach composing code with a focus on experimenting in JavaScript, a popular high-level scripting language used in thousands of websites and web applications. In fact, it is incredibly likely that most, if not all, of the websites you frequently visit rely on JavaScript. Because JavaScript can be written and interpreted in a web browser, there is no need to use a specific operating system or to install extra software in order to practice any of the exercises discussed in this chapter. While there are powerful and complex programs written in JavaScript that are run independently of a browser (such as Node.js, a platform for web applications), such programs are beyond the scope of our exercises here. The concepts covered in this chapter should provide robust enough material for an initial engagement with programming.

That said, although this chapter provides some introductory exposure to JavaScript and to computer code fundamentals, it is not *really* a useful substitute for a programming textbook; for those interested in learning more about code and JavaScript in particular, at the time of this writing Haverbeke's (2015) *Eloquent JavaScript* and Duckett's (2014) *JavaScript and jQuery: Interactive Front-End Web Development* are especially useful and accessible for the novice programmer. For our purposes, it is enough to learn about the principles described in this chapter and to apply them toward certain rhetorical ends via the included exercises.

Procedural Progymnasmata

Classical Greek rhetoric introduced the concept of the *progymnasmata* to students of rhetoric seeking to improve their abilities, and the progymnasmata have become staples of rhetorical education since. Progymnasmata are exercises in training particular rhetorical principles in order to improve a student's use of those being emphasized in any particular example.

In many ways, progymnasmata encourage imitation of particular strategies as forms of invention by constraining students to focus on a small range of questions or considerations during a single exercise. As Crowley and Hawhee (2009) have described it,

> Imitation exercises, if practiced in the way that the ancients practiced them, can lead to a more finely tuned rhetorical method of reading and listening. That is, when reading and listening rhetorically, we read and listen as much for *how* a writer or speaker builds an argument with words, sentences, paragraphs, and sections, as for *what* the writer or speaking is arguing. (29)

The exercises presented in this chapter definitely emphasize this focus on *how* a particular exercise is completed, as well as *why* it is completed in a given fashion. One of the central goals is to illuminate how procedural rhetoric can be developed and delivered to various readers rather than simply received from some other agent. Some of these exercises may have clearer purposes or more easily accessible means of achieving a given outcome. Some of them are certainly simpler or more complex than others. They all work rhetorically through blocks of code in words and often

resemble (but are distinct from) "sentences, paragraphs, and sections" (Crowley and Hawhee 2009, 29), but they are as inherently rhetorical as any conventional text.

While it is highly unlikely that a reader unfamiliar with programming will complete this chapter feeling as though he or she has gained full coding proficiency or code literacy, there is nonetheless much to be explored and considered regarding how even seemingly trivial decisions about individual procedural operations can have a significant impact not only on how a program is executed or interpreted by a computer but also on how it is understood to operate by a human reader. In turn, these considerations can lead to more nuanced and comprehensive approaches to teaching programming as a rhetorically informed composing activity.

Writing with Procedure

Computers function, fundamentally, on a very simple logical scheme called Boolean logic (discussed in chapter 2). Boolean logic assumes that all operations can be reduced to binary data, such as yes/no, on/off, true/false, 1/0. Fortunately, it is possible to develop far more complex procedures and computations that make use of combinatoric Boolean calculations. As some exercises below will demonstrate, a common means of developing a program involves composing conditional statements that check the status of one or more data points in order to express some particular set of operations; a readable example might involve checking the day of the week in order to display the proper planning schedule. But before we turn to these logical structures and the nuances of conditions, we first want to make sure that our programming environment is in a suitable working order.

Perhaps the single most common initial program for novice developers, regardless of specific programming language being learned, is generally called "Hello World." The program's name derives from the text output provided when it is run in order to show its writer that the program has been successfully executed. A "Hello World" program often has two components: first, a call to some command or function that will output text; second, the text itself, that is, the "hello world!" message. As you might guess, the syntax for each language looks either trivially or significantly different from that of other languages, even for a program as minor as this one.

For JavaScript, the "Hello World" program typically looks like the following one-line statement, made up of one function and a single parameter to be passed through it:

```
alert("hello, world!");
```

What the above line of code effectively says is for the interpreter to render an "alert"—that is, to display a pop-up window—with the quoted text **`("hello, world!")`** within the **`alert()`** function's parentheses. The semicolon at the end of the line tells the interpreter that the statement has concluded, much like a period at the end of a sentence. One could replace the text **`hello, world!`** with any other text, and the program *should* work just as smoothly, so long as the input text is surrounded by quotation marks. As a brief aside: the remaining exercises will not use the **`alert()`** function, instead displaying output within a JavaScript console environment. The pop-up window that **`alert()`** creates is a helpful visual indicator that the code has successfully been interpreted.

A program like "Hello World" provides the fundamental components of a language's syntax to begin orienting a novice programmer to the act of composing in code. There's a *tremendous* amount still to learn, but this first step is meant to make everything else just a bit easier and more familiar for future experiments in programming.

In order to make any program work, however, we need to understand some essential concepts related to Boolean logic, as this is what allows computers to function. Boolean logic is a logical system in which everything can be reduced to a binary: true or false, yes or no, positive or negative, 1 or 0. Computer circuits make use of electron charges to parse computational operations, with powerful circuits employing hundreds or thousands of "logic gates" to determine the answers to specific calculations, using similar binary calculations to determine relations among individual points of data, for example, a AND b (output "true" if both variables' values are "true"), a OR b (output "true" if either is "true"), a NOR b (output "true" if both variables' values are "false"), a NAND b (output "true" if *only* one of the two is "true"), etc. While none of the exercises in this chapter will tax a JavaScript interpreter, it is important to realize that the logical considerations we will make here work as microcosmic reflections of far more robust and important programs; there's nothing *fundamentally* different about the underlying nature of any software. We'll just be working at a much, much smaller scale in the following exercises.

Learning JavaScript Syntax

Like any language, JavaScript has a number of syntactic and vocabulary-related idiosyncrasies and nuances one must know in order to make effective use of the language. Many of these idiosyncrasies resemble those of other, related languages. Some are unique to JavaScript. In either case, recognizing the flexibility that *any* programming language has when it comes to names, syntax, and so on is an important step in becoming capable of employing that language for particular purposes. (Hopefully, this basic concept sounds familiar to any rhetorician.) Almost all software code works in combinatoric fashion, with discrete statements being combined together to form more complex computations of data. The following two-line program serves as a basic example of combinatory logic. First, we define a variable which has been named **`myVariable`** and is provided a Boolean value of **`true`**. Second, we display a message that includes the current value of said variable.

Practice Script 5.1: Simple statement combination

Line	Code
1	`var myVariable = true;`
2	`"The value of myVariable is " + myVariable;`

Admittedly, there is not much to this program, but it does at least illuminate how we can see individual statements referring to—and in many cases relying on—other statements in order to create more complex calculations and manipulations of data. This combinatorial approach is much more impressive when we want to make use of a *lot* of data values together, such as when we might want to display a list of volunteer names and relevant contact information. We will see this sort of computational power play out through the use of loops that perform similar (if not identical) calculations on each member of a set of data. Before turning to loops, however, it is important to examine how conditional statements work.

Conditional statements make use of Boolean logic to execute particular operations when certain parameters (conditions) apply. For example, we could revise our two-line program (in practice script 5.1) to incorporate a single condition so as to display a message only when that condition is met, as demonstrated in practice script 5.2. In this program, the condition begins on line 2 and ends on line 4. Due to its multiple-line scope, the conditional operations are incorporated within a pair of curly brack-

ets, which denote their subordinate relationship to the condition in which they reside.

Practice Script 5.2: Conditional statement syntax

Line	Code
1	`myVariable = true;`
2	`if (myVariable == true) {`
3	`  "The value of myVariable is " + myVariable;`
4	`}`

Combinatorially, there are several kinds of statements occurring in the above lines of code, and each one performs a very different function than the others. After the variable is defined in the first line, the second line sets up a conditional statement that checks the value of a given calculation,in this case comparing the value of **`myVariable`** to the expected value (**`true`**) provided after the two equal signs. (On a related note: line 2 could be written with Boolean shorthand as: **`if (myVariable) {`** .) The brackets, opened at the end of line 2 and closed on line 4, serve as containment markers for statements included between (within) those brackets. What this means for our purposes is that any subordinate statements, those provided within the brackets, will only be computed when the conditional calculation is met, that is, when **`myVariable`** is set to **`true`**. The statement on line 3 simply prints out a message that confirms the expected value of **`myVariable`**. Of course, if the variable is ever given a different value, then there will be no output message displayed. In its current form, the program's second condition—when **`myVariable`** is NOT set to **`true`**, that is, when it is **`false`**—remains only implicitly included as a consideration.

Thus, we could write a second statement that only appears when **`myVariable`** is **`false`**; this would require a second condition to be added to the program. Unlike our first **`if()`** statement, this second condition could be anchored to the first, so as to suggest to any readers that a relationship exists between the multiple condition checks we are making. Specifically, in this case, it is a relationship of exclusive distinction: if the first condition is not met, then the program executes any checks for remaining conditional parameters. Here is an example of the expanded program:

Practice Script 5.3: Revised simple statement combination

```
Line   Code
1      myVariable = true;
2      if (myVariable == true) {
3        "The value of myVariable is TRUE";
4      } else {
5        "The value of myVariable is FALSE";
6      }
```

With these added lines, several additional operations occur. The first—**`else()`**—means that its subordinate statements will only be computed whenever the original **`if()`** condition is *not* met, that is, when **`myVariable`** is **`false`**. A similar output message is provided so that we know the current value of **`myVariable`** (not that anything has been done to change it).

But, while these initial programs should successfully execute, neither of them is particularly *elegant*, that is, the purpose for these programs is quite simple, and *for this purpose* the provided code is not as clear or concise as it could be, given how JavaScript works. Our initial program (in practice script 5.1) does its job much more concisely, but the conditional logic demonstrated in practice script 5.2 and practice script 5.3 allow glimpses into much more complex and elaborate computational statements, as we will see shortly in combination with loops that allow for iteration through sets of data.

Exercises in Repetition: Looping

As mentioned in the previous chapter, repetition can be an incredibly powerful tool for programmers, provided that the repetition under discussion works in support of broad principles relating to conciseness, clarity, readability, and computational elegance. This holds true even if the *output* of a program is otherwise identical; for the audience of developer readers, code is generally less clear and comprehensive in its expression than it is in its source text.

In order to make use of numerous and multiple computations of data, programmers will frequently make use of *loops* to iterate through their intended data sets. Loops facilitate concise composition of potentially complex calculations that do not need to be entirely unique from other, similar calculations. Just as we do not "reinvent the wheel" by developing entirely new processes when the tasks in which we regularly engage are altered

slightly (e.g., driving a rental car instead of one's own car), neither do programmers generally seek to create new algorithms for variations on existing ones.

The example program in practice script 5.4 is certainly longer than those provided so far in this chapter, but that length is meant to reflect the difference between looping and *not* looping through similar kinds of data points. Those with programming experience will recognize it as an incredibly inefficient program that is effectively impossible to maintain or scale (such as if we wanted to display not letters of the alphabet but some other set of data with multiple elements).

Practice Script 5.4: Non-looping iteration through the alphabet

Line	Code
1	`output = "";`
2	`output += "a";`
3	`output += "b";`
4	`output += "c";`
5	`output += "d";`
6	`output += "e";`
7	`output += "f";`
8	`output += "g";`
9	`output += "h";`
10	`output += "i";`
11	`output += "j";`
12	`output += "k";`
13	`output += "l";`
14	`output += "m";`
15	`output += "n";`
16	`output += "o";`
17	`output += "p";`
18	`output += "q";`
19	`output += "r";`
20	`output += "s";`
21	`output += "t";`
22	`output += "u";`
23	`output += "v";`
24	`output += "w";`
25	`output += "x";`
26	`output += "y";`
27	`output += "z";`
28	`output;`

As should be quickly evident, the program in practice script 5.4 is incredibly tedious to write, even if it is quite readable. The program performs *almost* the same task twenty-six times, with the output message (each letter of the alphabet) as the only variable component involved in the program's execution. Replacing this repetition with a loop not only makes the program considerably shorter but also provides a shorthand description of the nearly identical task to be completed in such a way as to help the programmer audience understand how the task applies similarly to each data point. Practice script 5.5 is a revised version of the program in practice script 5.4. In this revision, the program makes use of a data type called an array, which contains multiple elements that can be called by their position within the overall array (starting from zero); that is, the value of **`alphabet[2]`** is **`c`**. Then, the program makes use of a loop to iterate through each of the array's elements and append its value to another variable, **`output`**, the full contents of which are displayed once the loop is completed.

Practice Script 5.5: Looping iteration through the alphabet

Line	Code
1	`alphabet = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"];`
2	`output = "";`
3	`for (i=0;i<alphabet.length;i++) {`
4	`  output += alphabet[i];`
5	`}`
6	`output;`

In this looped example (practice script 5.5), depending on your familiarity with iteration, the program is either much easier or much more difficult to read than the nonlooping version. To the user running this program, the same output is displayed regardless of which version of the program is provided.

JavaScript syntax for loop parameters involves three components. The first is to define a variable (in this case, **`i`**) with an initial value that will be modified over the course of the looping. The second component describes the condition(s) in which to continue iterating through the loop (in this case, so long as the value of **`i`** is less than the *length* of **`alphabet`**, which refers to the number of elements—here, single-character strings of text—in the array variable). The third component describes what action

to perform upon each iteration (**`i++`** is shorthand for "add one to the current value of **`i`**"). Technically, there is no *inherent* relationship between the length of the **`alphabet`** variable and the **`i`** counter variable: this particular program defines them as having the same "size" and thus the loop will occur the same number of times as there are elements in the **`alphabet`** array. The loop's condition could be altered to provide some other boundary value; this might (if the number is less than 26) result in displaying only some of the letters or (if the number is greater than 26) displaying an error message.

Similarly, we could change the order of the displayed elements, as in practice script 5.6, where the values of **`alphabet`** are displayed in reverse order; the loop—rather than starting from zero (the array's initial element position) and working toward the end—begins with the final element position and works back to the first.

Practice Script 5.6: Looping iteration backwards through the alphabet

Line	Code
1	`alphabet = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"];`
2	`output = "";`
3	`for (i=alphabet.length-1;i>=0;i--) {`
4	`  output += alphabet[i];`
5	`}`
6	`output;`

We could also, however, randomize our various calculations to generate less predictable output. The following revision of our program makes use of several built-in JavaScript methods that will generate a random number for us (specifically, **`Math.random()`**, which returns a floating number between 0 and 1) and then make said number into an integer (a combination of the method **`Math.floor()`** and a multiplication of the randomized floating number), so that we can successfully call one of the array elements to be displayed.

The revised program in practice script 5.7 performs the same *basic* function as the previous versions—printing elements from the **`alphabet`** array—but we can no longer predict precisely which element will be displayed at any moment. That said, there is still a described relationship between the loop and its output: the range of potential elements is influenced by the current value of the iterating **`i`** variable (e.g., the first

Practice Script 5.7: Looping iteration through a randomized set of alphabet elements

```
Line  Code
1     alphabet = ["a", "b", "c", "d", "e", "f", "g", "h", "i",
      "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u",
      "v", "w", "x", "y", "z"];
2     output = "";
3     for (i=0;i<Math.floor(Math.random() * alphabet.length);i++)
      {
4       output += alphabet[Math.floor(Math.random() * alphabet.
      length)];
5     }
6     output;
```

displayed value will not always be **a**, even though **i** initially equals zero, as established in line 3). Unlike in the previous two versions of this program, though, the arbitrary nature of the relationship between "iterator" and "data being iterated" has been clarified through the randomization of both the loop and its displayed output.

Exercises in Style: FizzBuzz

Among the most common "simple" looping programs written is "FizzBuzz," often as part of an initial programming job interview. The purpose of "FizzBuzz" is ostensibly to take the numbers 1 through 100 and print each of them out in order, unless a number is a multiple of three, in which case it is replaced with the word "Fizz," or if it is a multiple of five, in which case it is replaced with the word "Buzz." While "FizzBuzz" serves in part as a quick means of determining whether an applicant possesses fundamental knowledge about programming, it also—as described briefly in chapter 2—illuminates some important rhetorical information about the way a given developer approaches relevant programming tasks. We can gain a sense of how the author imagines each operation of the loop being executed in accordance with a particular logic. We can recognize how the author perceives relationships between various components of the loop. We can comprehend basic considerations of readability for (what are usually) simple statements.

This is not to suggest that "FizzBuzz" should be seen as having some incredible significance as a program; it has a limited scope and there is only so much we can learn from such a genre. But we *can* learn from it,

and this information is primarily only available by examining the source code for "FizzBuzz" rather than its output data, which optimally should be identical to that of any other version of the program.

Practice script 5.8 and practice script 5.9 provide two versions of "FizzBuzz" written in JavaScript, slightly different from those originally provided in chapter 2 as examples within table 2.2. If we run both of these two versions, we can see identical output, but the logic of the two loops is quite different. Practice script 5.8 allows for inclusive conditional checks; that is, output could be provided whether one or two conditions is met. In contrast, practice script 5.9 employs exclusive conditional logic, meaning that only one output message will ever be displayed in response to a given iteration of the loop. The ordering of the conditions in practice script 5.9 is also significant: the condition for **`i%15`** (meaning "get the remainder, or modulo, of **`i`** divided by 15") is provided first, since multiples of 3 or 5 would otherwise trigger that output and thus create an "incorrect" version of "FizzBuzz" in regards to its anticipated operation.

Despite their differences in logic, these two programs do not encompass the range of approaches—styles—by which one might write a "FizzBuzz" program in JavaScript. Other versions might rework the arrangement of the conditions used, others might employ entirely different conditions altogether, others might or might not frame the loop within the bounds of a **`function()`** like that these did so that the loop can be executed again easily, and others still may emulate the syntax or style con-

Practice Script 5.8: "FizzBuzz" with inclusive conditional loops

Line	Code
1	`function iterate() {`
2	`  for(i=1;i<=100;i++) {`
3	`    if ((i%3==0) || (i%5==0)) {`
4	`      if (i%3==0) {`
5	`        output += "Fizz";`
6	`      }`
7	`      if (i%5==0) {`
8	`        output += "Buzz";`
9	`      }`
10	`    } else {`
11	`    output += i;`
12	`    }`
13	`  }`
14	`  return output;`
15	`}`
16	`iterate();`

Practice Script 5.9: "FizzBuzz" with exclusive conditional loops

```
Line  Code
1     function iterate() {
2       for(var i=1;i<=100;i++) {
3         if (i%15==0) {
4         output += "FizzBuzz";
5         } else if (i%3==0) {
6           output += "Fizz";
7         } else if (i%5==0) {
8           output += "Buzz";
9         } else {
10          output += i;
11        }
12      }
13      return output;
14    }
15    iterate();
```

ventions of other languages with which they are more familiar. As a genre, "FizzBuzz" offers a number of opportunities not only for rhetorical analysis (e.g., of existing versions of the program) but also for inventive possibility when writing new versions of the program. This is not to suggest that the program must always be rewritten in novel ways, just that it *can* be, and attempting to work through how to write such a program can serve as a useful exercise for us. The following examples of potential "FizzBuzz" programs attempt to provide identical output to those in practice scripts 5.8 and 5.9, but they do so with trivial or significant differences in how that output is generated.

The example in practice script 5.10 is a particularly inefficient means of executing the "FizzBuzz" loop. Thanks to an array-specific method (**`forEach()`**), the program does technically iterate through a set of data—the elements of the **`myArray`** variable—but the "real" computational work of the program is hard-coded into the array elements. That is, the program does not determine when to print a number or a string, but instead prints the values entered directly by the author. There is no "discovery" here of how the program will calculate each of its data points; the output simply repeats the contents of the array in order. The resulting output remains identical to practice script 5.8 and practice script 5.9, but the source code of practice script 5.10 betrays its lack of any conventional elegance (whether using computational or stylistic definitions of elegance). Despite its lack of elegance, the program nonetheless offers useful insight

into how an author *could* compose a solution that hinges on a particular understanding of the program's needs (a clear data set and the need to display each item within that set). That the program cannot be easily modified, such as in terms of altering which data points are displayed as numbers or strings of text, is of incidental concern.

Practice Script 5.10: "FizzBuzz" with static array elements

```
Line  Code
1     myArray = [1, 2, "Fizz", 4, "Buzz", "Fizz", 7, 8, "Fizz",
      "Buzz", 11, "Fizz", 13, 14, "FizzBuzz", 16, 17, "Fizz", 19,
      "Buzz", "Fizz", 22, 23, "Fizz", "Buzz", 26, "Fizz", 28, 29,
      "FizzBuzz", 31, 32, "Fizz", 34, "Buzz", "Fizz", 37, 38,
      "Fizz", "Buzz", 41, "Fizz", 43, 44, "FizzBuzz", 46, 47,
      "Fizz", 49, "Buzz", "Fizz", 52, 53, "Fizz", "Buzz", 56,
      "Fizz", 58, 59, "FizzBuzz", 61, 62, "Fizz", 64, "Buzz",
      "Fizz", 67, 68, "Fizz", "Buzz", 71, "Fizz", 73, 74,
      "FizzBuzz", 76, 77, "Fizz", 79, "Buzz", "Fizz", 82, 83,
      "Fizz", "Buzz", 86, "Fizz", 88, 89, "FizzBuzz", 91, 92,
      "Fizz", 94, "Buzz", "Fizz", 97, 98, "Fizz", "Buzz"];
2     output = "";
3     myArray.forEach(function(i) {
4       output += i;
5     });
6     output;
```

In contrast, practice script 5.11 makes use of multiple variables that allow for the test program to be adjusted as desired beyond the loop bounds of 1–100 (with a potentially different scope of iteration), for new conditions to be checked, or for different output to be displayed. The basic exclusive conditional logic used in practice script 5.11 is *nearly* identical to that of practice script 5.9, save that the exclusive nature of each check must be more explicit, as the relationship between **`multipleA`**, **`multipleB`**, and **`multipleC`**—should the parameters for this "FizzBuzz" program ever change—may not always be easily recognizable (if such a relationship exists at all). Specifically, each of these conditions employs Boolean logic to rule out multiple successful checks, so line 11 describes checking whether **`i%multipleA`** equals the value of **`checkedMod`**, so long as the same is NOT also true for **`i%multipleB`** and **`i%multipleC`**.

Practice script 5.11 is the longest of the looping versions of "FizzBuzz" provided in this chapter, but its length does not make it necessarily unwieldy or unreadable. Due to its "scalability first" approach, however, it does suggest a very different programming philosophy than those in prac-

Practice Script 5.11: Modular/scalable "FizzBuzz" program

Line	Code
1	`checkedMod = 0;`
2	`multipleA = 3;`
3	`multipleB = 5;`
4	`multipleC = 15;`
5	`messageA = "Fizz";`
6	`messageB = "Buzz";`
7	`messageC = "FizzBuzz";`
8	`function iterate(minTotal, maxTotal, iterateAmount) {`
9	`  for (i = minTotal; i <= maxTotal; i+= iterateAmount) {`
10	`    if ((i%multipleA == checkedMod) && (i%multipleB != checkedMod) && (i%multipleC != checkedMod)) {`
11	`      output += messageA;`
12	`    } else if ((i%multipleB == checkedMod) && (i%multipleC != checkedMod)) {`
13	`      output += messageB;`
14	`    } else if (i%multipleC == checkedMod) {`
15	`      output += messageC;`
16	`    } else {`
17	`      output += i;`
18	`    }`
19	`  }`
20	`  return output;`
21	`}`
22	`iterate(1, 100, 1);`

tice scripts 5.8, 5.9, or 5.10, namely, that a program can and should be written in such a way as to reduce any potential redundancy in future revisions of that program. The passing of custom input data to the function each time it is called allows for experimentation with the "FizzBuzz" algorithm. Several input data points are established as passing to the function in line 8 of practice script 5.11, with an example call of the function in line 22. While the majority of these "FizzBuzz" programs *work*, they are not so easily adjustable for alternate data (that is, the loops themselves would be rewritten rather than the values for individual variables).

As progymnasmata, what these "FizzBuzz" examples hopefully demonstrate, more than anything else, is the power of rhetorical style in establishing an argument. It illuminates particular logics and decisions behind a rhetor's claims, even if the ultimate "point" may seem indistinguishable from that of other arguments. While we often consider issues of style and arrangement with an orientation toward audience reception and subse-

quent action, we can attend equally to the means and meaningful mechanisms by which a rhetor has established a particular line of reasoning or how that rhetor has framed significant concepts or terms as a reflection of his or her own comprehension of the issue at hand.

Exercises in Repetition: Object Creation

Just as iteration is an important component of computation and thus computational rhetoric, so too is repetition—explicit and implicit—central to many programming languages, especially object-oriented languages that rely on class inheritance. "Class inheritance," as described in earlier chapters, refers to the way individual objects (bundles of data) are defined by default and how they are capable of behaving through the employment of general and specific computational procedures. Developers rely on class inheritance principles for iterative object creation and modification in order to avoid potentially overwhelming amounts of repeated code throughout their programs.

JavaScript, however, is not *technically* an object-oriented programming (OOP) language. Instead, JavaScript is "prototype-based," meaning that objects and object types have properties linked from their prototypes rather than having properties copied (inherited) from classes; even so, much of the language's behavior emulates that of OOP languages (deasydoesit et al. 2018). It is possible, however, to define and develop object classes for use in individual programs; further, a number of standard object types are built in to JavaScript by default, including **`Math`**, **`String`**, **`Array`**, and **`Object`**. As a result, JavaScript functions flexibly for programmers who both do and do not want the language to adhere strictly to OOP principles.

Thanks to JavaScript's flexibility, we are offered several opportunities for rhetorical experimentation here. First, we can play with anticipation through the development of a class, establishing attributes and methods that will likely be used by any future instance of said class. Second, we can examine repetition as it occurs through modification, of changes, across instances of a concept, in data values and computational behavior. Third, we have the potential to see how changes, whether significant or seemingly trivial, can have striking effects on the arguments we make in and through the code we write.

The code in practice script 5.12, inspired by the JavaScript example in table 4.5 by saskatchewancatch et al. (2016), creates a class (called **`Cy-`**

`borg`) and then creates two objects that inherit the class's defined behaviors. Functionally, the program is quite simple, as the only behavior defined initially is the establishment of a **`name`** attribute, based on the **`myName`** argument passed to the class function when each new object is created. Through the composition of this program, however, we can get an initial sense of how effects of the work performed in developing a class echo through the objects we create from that class. In this case, line 4 returns text confirming each new object's creation with a message that includes its custom **`name`** data when the **`greet()`** method is called, as in line 8. While this is not a complicated operation, it nonetheless hinges on the expectation that each new relevant object will have **`name`** data set upon its initialization (in part so that the message will appear to the user as intended). Any **`Cyborg`**-based objects without this variable are thus missing a potentially vital component to their "existence" in the program and its purpose.

Practice Script 5.12: Simple class construction and object initialization

```
Line   Code
1      Cyborg = function(myName) {
2        this.name = myName;
3        this.greet = function() {
4          return "New cyborg, " + this.name + ", activated.";
5        }
6      }
7      myCyborg = new Cyborg("Version 1");
8      myCyborg.greet();
```

To the nonprogrammer user, if another object using this class is created with *no* name data, almost everything will function "correctly." As it stands, the only visible proof of some issue or bug with the program, outside the bounds of the source code itself, would be a blank space in the middle of the message outlined in line 4, specifically, where **`this.name`** is called and expected to be populated with its current value (set in line 2). Not every space for error or unanticipated behavior (e.g., not planning for an object to be created *without* a "name," although it is possible to do so) allows for the otherwise successful execution of a program, and the attention paid to such concerns reflects the values of the program's authors as much as does any explicit line of code. Unfortunately, these values may not always be shared by or accessible to users of the program.

Practice script 5.13 provides more complexity in describing how the

Cyborg class and its resulting objects operate, although fundamentally the program still makes use of simple string data stored in variables. Its capabilities have expanded in regards to the use of data that is dynamically generated when particular methods are called. In addition, the program employs *chaining* and *nesting* to accomplish complex goals via combinations of individual methods with narrow purposes. In effect, the program's code anticipates (through its structure and logic) significant and varied forms of computational repetition to be encountered when it is run.

Chaining and nesting refer to distinct but related concepts in which specific procedures are called by other procedures in order to accomplish a different—generally more complex or comprehensive—goal than that

Practice Script 5.13: More complex class and object creation

Line	Code
1	`function Person(firstName) {`
2	`  this.firstName = firstName;`
3	`}`
4	`function Machine(serial) {`
5	`  this.serial = serial;`
6	`}`
7	`Cyborg = function(firstName,serial) {`
8	`  Person.call(this,firstName);`
9	`  Machine.call(this,serial);`
10	`  name = firstName + "-" + serial;`
11	`  birthdate = Date();`
12	`};`
13	`cyborg1 = new Cyborg("Grover", "1");`
14	`Cyborg.prototype.greet = function() {`
15	`  return "Hello world, I am " + name + "! I was created on " + birthdate + ". Despite my first-person statements, I am not quite self-aware. ";`
16	`  this.ageUpdate();`
17	`}`
18	`Cyborg.prototype.ageUpdate = function() {`
19	`  age = Date.parse(Date())-Date.parse(birthdate);`
20	`  ageSec = (age / 1000) % 60;`
21	`  ageMin = (age / 60000) % 60;`
22	`  ageHrs = (age / 3600000) % 24;`
23	`  ageDays = (age / 86400000);`
24	`  return "I am " + Math.floor(ageDays) + " days, " + Math.floor(ageHrs) + " hours, " + Math.floor(ageMin) + " minutes, and " + Math.floor(ageSec) + " seconds old. ";`
25	`}`
26	`cyborg1.greet();`

of any individual component. Chaining refers to the practice of constructing new procedures by calling multiple methods at once (thus building a "chain") without relying on variables to store the results of each individual method. For example, line 20 involves the use of two **`Date.parse()`** methods and a mathematical operation to calculate the value of a new variable called **`age`**, which is then used as the basis for other calculations. Rhetorically, chaining is not unlike writing a statement or series of statements involving multiple instances of jargon: it might be *possible* to frame the statement(s) in such a way as to incorporate definitions of each term, but it is *probable* that any terms are explained elsewhere so as to improve readability and relative conciseness of statements using those terms.

Nesting, meanwhile, refers to the use of procedures within (or subordinate to) other procedures in order to complete a given task. Since some tasks may be common across a set of important procedures, the tasks may be modularized as methods and then called as desired without the need for them to be rewritten for each larger procedure in which they appear. Line 17, for example, nests the **`ageUpdate()`** method within the **`greet()`** method; **`ageUpdate()`** exists and can be called independently, but its functionality is *also* an expected or desired component of the operations comprising **`greet()`**. A potentially useful analogy from classical rhetoric is the syllogism, in which premises are already accepted in order for one or more conclusions to be reached. One or more premises may be debatable, but the resolution of that debate generally exists *outside* the syllogism; its presence *within* the syllogism serves to facilitate the completion of the larger argument. As part of completing the syllogism (i.e., following its logic), one might question the strength or veracity of a premise, but that process occurs internally as a kind of subroutine necessary to understand and deliberate on the larger conclusion(s) at hand.

The program's use of dynamically generated data further contributes to its complexity. No longer can we assume that the same output will always be displayed when we call for it; instead, the program's code takes on a more (rhetorically) active role in its expression by generating data not explicitly and statically entered by the program's author (who may not necessarily know or, in some cases, even accurately guess at what the generated data may be). This means that *composing* such a program inherently involves anticipation of potentially diverse expressions emerging from its use—what rhetoricians would recognize as multiple and different rhetorical contexts—and the skilled developer must recognize at least some of the bounds of that diversity constructed within their code. Admittedly,

the "diversity" relating to this program is relatively limited, as the "age" of any **`Cyborg`** object created will always increase, its output will always be displayed as a set of specific numbers, and it is possible to guess with some accuracy what any given **`greet()`** message will display. As an initial exercise in creating objects that make use of dynamic data, however, this program offers a far greater range of possibilities than those in the previous progymnasmata.

Exercises in Arrangement: Bubble Sort

Modularity, as described earlier in this chapter, is an important component of many programming languages, since it allows for a decrease in explicitly repeated code. An even more important component for programming, and for procedure in general, is order, that is, the arrangement of particular procedural operations in the order necessary to produce a desired result. This identification of order as important is likely obvious, given the examples already provided that make use of particular procedural structures to induce certain outcomes. Similarly, the rhetorical canon of arrangement refers to the skillful construction of arguments that promote particular claims and offer proofs supporting those claims.

Fundamentally, there is little distinction between the arrangement of arguments composed in conventional discourse and those composed in code, save perhaps for the means by which important terms and concepts are established. Specifically, conventional rhetors anticipate reaching a sense of agreement or *stasis* with audiences about multiple facets of their arguments (e.g., agreement on questions like the following: Was something done? What was done? Was it proper or appropriate to be done?). In the majority of programming languages, where statements are overwhelmingly imperative declarations, the notion of stasis differs; a given programmer—within the constraints of a given language and a context of hardware and software limitations—outlines the boundaries for the *logos* that drives the program the programmer has written. These boundaries can include establishing a particular variable as a certain type of data (e.g., an integer or a string of text), affording an object only certain attributes or behaviors (e.g., the earlier exercise's **`Cyborg`** output messages), and even constructing procedural behaviors themselves.

As a result, arrangement takes on an entirely new dimension when understanding programming as a rhetorical activity. If a programmer can rely entirely on the constraints of a language to perform the

work of definitional suasion, then the means by which a given concept is communicated—which, for code, is the structure and order of one or more relevant procedures—becomes that much more central and significant to its effectiveness for human as well as machine audiences. Obstacles to readability, computational efficiency, or scaffolding of calculations can all lead to drastically different outcomes than those potentially intended by a program's author(s).

One type of programming exercise that emphasizes procedural arrangement is the "Bubble Sort," which refers to a program that takes a set (usually a list) of data points and reorders them according to a particular system (such as arranging words in alphabetical order or arranging numbers from smallest to largest). The name "Bubble Sort" refers to the way certain items in the set move toward the end of the set, resembling the rise of bubbles in a liquid. The "Bubble Sort" program employs several complementary procedural mechanics, including nested loops and multiple operations relating to the storage and manipulation of data points within a larger record. It is important for a programmer to comprehend how the different procedures work in conjunction with one another and that they do so in a precisely arranged fashion so as to effect the desired outcome.

As a related example, in her book *Exercises in Programming Style*, Lopes (2014) demonstrates 33 different stylistic approaches to a computational task known as "term frequency," which conceptually resembles the "bubble sort" procedure with some additional data sorted and displayed to the user. Lopes defined the "term frequency" task as a trivial one: "[T]he computational task in this book is trivial: given a text file, we want to produce the list of words in the file and their frequencies, and print them out in decreasing order of frequency" (2014, xiii). The procedure can certainly be described simply, and—depending on the programming philosophy employed to compose it—may be realized simply as well. However, as Lopes' book illuminates the significance of style, and as this exercise attempts to demonstrate regarding the arrangement of data and logic in two stylistic approaches, we can learn a great deal about a program as well as its author(s) when we examine *why* and *how* certain perspectives on procedural rhetoric are applied to the activity of composing this program (or other programs, of course).

The example "Bubble Sort" shown in practice script 5.14 employs two loops, one nested in the other, to successfully iterate through the data to be arranged properly (the Array **`myData`** in Line 1). Further, a conditional statement on line 6, within the innermost loop, initializes manipulation

of Array data via a temporary variable that momentarily stores the original value of each element as it gets altered in the course of the loop.

Practice Script 5.14: Simple "Bubble Sort" program

```
Line   Code
1      myData = [2, 5, 9, 6, 3, 1, 8, 4, 7];
2      function mySort(input)
3      {
4        for (j = input.length-1;j >= 0; j--) {
5          for(i=0;i < input.length-1; i++) {
6            if (input[i] > input[i+1]) {
7              tempData = input[i];
8              input[i] = input[i+1];
9              input[i+1] = tempData;
10           }
11         }
12       }
13       return input;
14     }
15     mySort(myData);
```

This version of the "Bubble Sort" exercise can be useful as a relatively transparent demonstration of one logic system applied to the act of sorting. After all, there are several "moving parts" (figuratively speaking) that must work together for the data to be modified and reordered correctly. It may be useful to compare this sort of program to a more recognizable rhetorical example: the relationship between major and minor premises within an enthymeme or syllogism. While it may be possible to use either premise independently for other purposes, it is necessary that an audience understand the significance of a rhetor's effort to connect the premises so as to successfully reach the desired conclusion(s) stemming from them.

"Bubble Sort" as a rhetorical exercise here, however, is made unnecessary in terms of the practical *need* for it in regards to JavaScript. That is, JavaScript possesses a native method for precisely the kind of automated arrangement effected above, and the method—called **`sort()`**—works for numbers or text strings, as demonstrated in lines 1 and 2 of practice script 5.15. The method can be chained with another method, **`reverse()`**, to provide a descending result (instead of the default ascending order), as demonstrated in line 3 of practice script 5.15. What the native methods obfuscate is the specific procedural logic involved in the sorting we, as programmer-composers, call on in this program as compared with that of the previous example, which allows for a more varied and nuanced ap-

proach to sorting should we desire a less conventional ordering system for our data.

Practice Script 5.15: "Bubble Sort" with JavaScript sort() method

```
Line    Code
1       myData = [2, 5, 9, 6, 3, 1, 8, 4, 7];
2       myData.sort();
3       myData.sort().reverse();
```

Clearly, the two approaches to "Bubble Sort" suggest very different logics, with one keyed in to the affordances of JavaScript (and thus "obscuring," to some readers, the means by which a list is sorted while clarifying it to others) and the other assuming a need to develop an explicit sorting apparatus for readers—and, perhaps, the author himself or herself—to understand how one can achieve a particular goal.

Exercises in Invention: enthymemeGenerator.js

We can combine the work performed thus far in the course of these exercises to develop even more nuanced and complex programs. Such programs can allow us to anticipate new and different rhetorical contexts than those initially established within the parameters of the previous exercises. In this exercise, we will focus on the complementary effects of exploratory and combinatoric invention as we create a program whose objects perform a number of meaningful writing-related tasks.

Specifically, this program will allow for the creation of distinct objects that can store, manipulate, and output data, which in this case will be focused on the creation of a short enthymemic puzzle or game. (I should note that the logic generated semirandomly by this program will not always be clear, so the displayed enthymemes should be considered within the logical "world" boundaries of their related premises and conclusions.) Generally speaking, the program is not overwhelmingly complex, but in relation to the previous exercises presented in this chapter, it serves as a suitable capstone to this initial investigation of programming-based progymnasmata.

First, it is important to consider what such a program requires mechanically. As initially envisioned, this program has several interconnected components or tasks with distinct purposes:

1. Definition of objects to be initialized, which involves:
    a) Definition of data lists that will serve as enthymeme elements
    b) Description of behaviors for sorting data list elements
    c) Description of behaviors for displaying generated arrangements of enthymemes
    d) Description of behaviors for storing generated enthymemes
2. Initialization of object(s)
3. Operation of object methods that generate specific enthymemes

None of the involved components is complicated in and of itself, but it nonetheless matters that each can be identified for its contribution to the goal of the overall program and so that we, as the program's authors, can attend to the relationships each has with the others in anticipation of users engaging the program.

Second, it is also important to remember that this program could be written in a variety of ways, and the approach taken below is hardly the only—or in some ways the best—means of accomplishing the goals described above. Curious readers may find it fruitful to experiment further with their own version of the program or to tweak the version provided here to understand more clearly how the differences in approach can communicate meaning through procedure in a variety of noteworthy ways. How might seemingly trivial, or clearly major, changes to individual procedures impact the range of expressions possible when running the program?

With these questions in mind, we can turn to application: How *might* such a program be composed in code? The version provided below employs the concepts outlined in this chapter's earlier exercises as well as introducing several new principles and methods (native to JavaScript) to the program. Hopefully, this sample version demonstrates a recognizable foundation on which to build while illuminating new and different efforts to achieve the program's goals.

This version of the program suggests several key concepts as integral to the generation of enthymemes. The logic used to generate a given enthymeme reflects a set of values and means of engaging the world (in this case, other potential programmers, the author's knowledge of the JavaScript language, and so on). The program's argument occurs enthymematically, with a number of implicit premises accompanying the explicit lines of code, including the following: this is one way to build an enthymeme; this is a preferable way to build an enthymeme; the enthymemes it generates are meaningful; the user for this program is interested in working through the logic of a given enthymeme.

Just as it is not a guarantee that a given reader of the code or user of the program will accept—or is interested in pursuing—any of the above premises, it is within the author's power to make one or more of these argumentative premises stronger or clearer so that a reader is more likely to recognize, understand, and accept them, beyond the assumed "acceptance" of constraint accompanying all software use (that is, the recognition of software- and hardware-based constraints on what is *possible* with a given program). Accordingly, the program code sets up several central parameters that anticipate its range of potential expressions: enthymemes consist of two sentences, the enthymemes have at least two points of thematic relation, each sentence grammatically is structured as subject–transitive verb–object, and—presumably—the logic of each enthymeme is accepted by the reader.

First, the example to be discussed is provided in practice script 5.16.

Practice Script 5.16: Enthymeme generator built on earlier object creation code

Line	Code
1	`noun = ["apple", "banana", "canteloupe"]; // Expand these vocabulary lists for more interesting results!`
2	`verb = ["act", "bellow", "cry"];`
3	`adjective = ["angry", "beaming", "cold"];`
4	`quantity = ["all", "few", "half"];`
5	`function Person(firstName) {`
6	`  this.firstName = firstName;`
7	`}`
8	`function Machine(serial) {`
9	`  this.serial = serial;`
10	`}`
11	`Cyborg = function(firstName,serial) {`
12	`  Person.call(this,firstName);`
13	`  Machine.call(this,serial);`
14	`  name = firstName + "-" + serial;`
15	`  birthdate = Date();`
16	`  return "New cyborg, " + name + ", activated.";`
17	`};`
18	`Cyborg.prototype.singularize = function(term) {`
19	`  iesLetters = "aeou";`
20	`  esLetters = "hosx";`
21	`  if (term.charAt(term.length-1) == "y") {`
22	`  if (iesLetters.indexOf(term.charAt(term.length-2)) == -1) {`
23	`      temp = "";`
24	`      for (i = 0; i < term.length-1; i++) {`
25	`        temp = temp + term.charAt(i);`
26	`      }`
27	`      term = temp + "ies";`
28	`    }`
29	`    else {`
30	`      term = term + "s";`

Practice Script 5.16: (continued)

```
      }
    }
    else if (esLetters.indexOf(term.charAt(term.length-1)) != -1) {
      term = term + "es";
    }
    else {
      term = term + "s";
    }
    return term;
  }
  Cyborg.prototype.randomize = function(list) {
    randomTerm = list[Math.floor(Math.random() * (list.length))];
    return randomTerm;
  }
  Cyborg.prototype.enthymemeGenerate = function() {
    amount = this.randomize(quantity);
    subject1 = this.randomize(noun);
    subject2 = this.randomize(noun);
    action = this.randomize(verb);
    directObject = this.randomize(noun);
    this.sentence("major", amount, subject1, subject2, action,
  directObject);
    this.sentence("minor", amount, subject1, subject2, action,
  directObject);
  return output;
  }
  Cyborg.prototype.sentence = function(type, amount, subject1,
  subject2, action, directObject) {
    if (type == "major") {
      myQuantity = amount;
      mySubject = this.singularize(subject1);
      myVerb = action;
      directObject = this.singularize(directObject);
      majorPremise = myQuantity + " " + mySubject + " " + myVerb +
  " " + directObject + ".";
      majorPremise = majorPremise[0].toUpperCase() + majorPremise.
  substring(1);
      output = majorPremise;
    } else if (type == "minor") {
      mySubject = subject2;
      myVerb = this.singularize(action);
      directObject = this.singularize(directObject);
      minorPremise = mySubject + " " + myVerb + " " + directObject
  + ".";
      minorPremise = minorPremise[0].toUpperCase() + minorPremise.
  substring(1);
      output += " " + minorPremise;
    }
  }
  cyborg1 = new Cyborg("Grover", "1");
  cyborg1.enthymemeGenerate();
```

Outside of the word lists in the first several lines of the program (each of which is purposefully short so as not to double the overall length of this text), much of the first quarter of the provided code may look especially familiar, since it involves a construction of the **`Cyborg`** object type so that readers can build on their program drafts from earlier exercises. That code is included first here to provide context for the code to come. Unlike many programming languages whose code texts are compiled and executed, JavaScript interprets its lines, so the order of composed procedures matters (that is, the program will not operate successfully unless we first establish variables that are referenced elsewhere before we can write the code that makes those references).

A new set of methods appended to the existing code lines provides the necessary functionality for generating the randomized enthymemes for this exercise. There are three distinct but connected methods—**`singularize()`**, **`randomize()`**, and **`sentence()`**—that are all activated when the "umbrella" method, **`enthymemeGenerate()`**, is called. In each of these methods, a particular procedure produces a key component of the enthymeme that is ultimately generated and displayed; it is worth noting that in its current form, the program creates a "categorical" enthymeme, meaning that the expected conclusion to be drawn is one that relates the subject of the minor premise to the subject of the major premise.

Of the integral functions used in this program, the **`randomize()`** method (lines 41–44) generates a random number, using **`Math.floor()`** and **`Math.random()`**, and then pulls the array element of that number from the appropriate list (as established when the method is called). Its nature as nonspecific to any particular type of term (noun, verb, etc.) means that it can be used to populate any grammatical component for these premise statements. The **`singularize()`** method (lines 18–40) similarly performs in a generic manner, although its name suggests its use is specific, as singularizing a noun often appears opposite to singularizing a verb; in fact, when this **`singularize()`** method is used for nouns, it actually pluralizes them by appending characters (e.g., "s" or "es") to words. The specific appended characters are determined based on the final characters of a given word, established in lines 19–20 and checked in the conditional statements beginning on line 21. Thus a word ending in "y" will have that "y" replaced with "ies" unless the letter *just before* "y" is included in the string on line 19 (the variable **`iesLetters`**), in which case the word will be altered to have an "s" after the "y" character. While

the method's name is arbitrary, there is nonetheless a clear implication for how a reader is expected to consider its function and purpose, and admittedly there is significant room for misinterpretation, in terms of its use for nouns, as a result.

In contrast, **`sentence()`** has a potentially more straightforward workflow that may be easier to follow despite its more complex use of method chaining to construct its output. Some of the data used is provided when the method is called (such as in the **`enthymemeGenerate()`** method, in lines 51–52), while others are computed based on that initial input, such as in regards to the "singularized" version of a given term or whether a major or minor premise is being constructed. At the end of each premise's condition, the appropriate variables are arranged and formatted, in a very particular way, into a sentence that is then displayed to the user.

As with the other exercises, any or all of these functions could be structured in a variety of ways for drastically different results, each demonstrating a different *logos* of how the program's author approaches understanding and communicating important information about the creation and interpretation of enthymemes. The sample program, in its current form, makes a particular argument about what an enthymeme is, how it works, and what a user—as well as a potential contributing developer—is expected to think about the use of enthymemes. (Admittedly, some of these expectations may not be particularly meaningful in the context of this brief exercise.) A program that further randomized the provided components of an enthymeme, such as a conclusion as well as a premise, might demand a very different set of functions to achieve that goal. Similarly, a program that generated different types of enthymemes, or even different approaches to phrasing enthymeme elements, might call for very different sets and arrangements of sentence components.

Ultimately, this exercise is as much an attempt at broad reflection on one's invention practices as it is the specific application of particular programming concepts to develop an enthymeme generator program. In any given text, how much do we lean on particular avenues of establishing meaning as opposed to seeking out new or alternate ways of making arguments? How do we attempt to help readers understand not only an intended point or goal but also the logic behind our decisions? How much do we identify certain principles to strive for, and how do we go about attempting to realize those goals? These questions and others emerge as a result of such an exercise, and they can serve as valuable heuristics for

reading the composing work we do in any medium, although, thanks to its inherently procedural qualities and hybrid reader/interpreter considerations, code may well offer novel dimensions for the responses we develop to our reflective questions.

Conclusions

The exercises presented in this chapter attempt to demonstrate a variety of key rhetorical concepts that inform composing practices across genres, modes, and media and reflect composing in code specifically. While programming as a form of meaning making involves a functional need to compose code in such a way as to be readable (interpretable) by the computer, it is nonetheless also a form of making and communicating meaning to human audiences (including potential collaborators or even some users). Accordingly, we can approach composing in code as a rhetorical activity, allowing us to experiment with the development of arguments in code just as we would in other media. In addition, we can potentially investigate more effectively how professional and amateur programmers compose their work with rhetorical strategies (un)consciously in mind.

To be clear, these exercises barely scratch the surface of the potential range of tactics a rhetor might employ in code or in another medium. Nonetheless they suggest what I hope is a suitable variety of considerations regarding the importance of treating code as an important and inherently rhetorical form of communication. If as rhetoricians we want to understand and inform the effective composition of meaning through code, or if as programmers (professional or amateur) we want to develop approaches or workflows to coding that are accessible as well as efficient, it behooves us to attend more closely to the relationships between the progymnasmata of classical rhetoric and the exercises of contemporary programming texts and hiring processes, as together they can tell us a great deal about the forms of meaning making that code authors and readers alike are expected to recognize, understand, and engage in particular ways for similarly particular purposes.

CHAPTER 6

Conclusions

I undertook this project to pursue a critical examination of the relationship between the rhetorical possibilities of algorithmic computation and the computational qualities of rhetoric, taking advantage of opportunities created by current popular interest in code. 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 enough merely 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 the potential action(s) that can be undertaken as a result of this knowledge. What does recognizing computation or code as rhetorical get us? How can we better proceed with investigating *and making* meaning in the twenty-first century if we attempt to inform practices of composing code with rhetoric? How might a more procedurally oriented or focused theory of rhetoric affect our approaches to knowledge creation and communication with networked technologies?

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 point from which rhetoricians can incorporate computational technologies more fully and naturally into the body of objects of serious rhetorical study and composition. Further, I hope that scholars and practitioners outside the field of rhetoric may also find this project valuable for improving education and practice relating to software development by considering what rhetoric has to offer computer science and professional (and, for that matter, amateur) programming activities.

The field I have described is not meant to serve as a means of answering the questions asked by those studying digital rhetoric or software, but instead as a space in which to engage issues emerging from parallel and convergent inquiries undertaken by scholars in these disciplines, as well as in disciplines that may not initially seem clearly related or impacted by the intersection of rhetoric, software, and code.

Rhetorical Code Studies Thus Far

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. 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, and in the past several years in particular there has been an increase in explicit cross-disciplinary acknowledgment or engagement among scholars in these disciplines. This boundary crossing is particularly notable in regards to rhetoric, given the continued expansion of our understanding of "digital rhetoric" as an area worthy of serious study. Most rhetoricians interested in digital media, however, remain 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 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 induce other developers to engage in specific styles of development, writing code that functions in particular and meaningful 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 influence their fellow contributors. In chapter 3, I examined the discourse of the development community for the Mozilla Firefox web browser. A massive and popular open source software program, Firefox has been collaboratively developed for seventeen 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 broad, even when considering the relative scope of discussion is "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 rhetorical 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 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 induce change with these strategies, their *use* nonetheless has implicit effects on the developer audiences who engage those texts and practices.

The programming-oriented progymnasmata serving as the focus for chapter 5 offer some initial, and hopefully useful, perspectives on the act of programming as rhetorically significant *and informed* composing. While those exercises do not offer a comprehensive engagement with the flexibility of programming or of basic rhetorical theory as a means of developing a complex program, they nonetheless demonstrate the possibilities available to scholars and professionals interested in rhetoric or in software development as a form of meaningful communication. This recognition of the act's inherent rhetoricity is meant to complement the preceding chapters' focus on analysis of existing texts and practices so that we, as a collected body of scholars and practitioners, can more effectively turn toward improving relevant rhetorical and programming pedagogy and further the push for computational and procedural literacy, broadly speaking.

My efforts here have been bolstered considerably by the spur of scholarship in the past several years that has brought together rhetoric and software in significant and interesting ways. Most notably, Brown's (2015) investigation of the "rhetorics of software" serves as the most currently comprehensive effort to connect rhetorical criticism with the study of software. Building on a foundation of rhetoric as well as software studies and media criticism, Brown has performed a set of insightful investigations into how we might, and could, understand questions of ethics and hospitality in software and software-mediated contexts. As Brown has argued, "Digital rhetoricians can and should be participating in discussions of computation, and they should do so both by bringing rhetorical theory to bear on software and by rethinking rhetorical theory in light of the unique attributes of computational media" (180). Similarly, Beck (2016) offered a number of questions pertinent to the rhetorical study of software and code:

> [H]ow might a rhetorical code studies treat social and cultural theories alongside non-human theories of machinic contexts? Additionally, how might focusing scholarly attention toward rhetorical and theoretical treatments of computer algorithms open interdisciplinary conversations and relationships? How might such perspectives attract complementary and divergent views? Since algorithms affect changes in machine and human behaviors, as the two scenarios that frame this article illustrate, how might those allied with rhetoric and writing studies gift a path toward greater knowledge about the formation, creation, and use of computer algorithms in myriad digital and scholarly spaces? (n.p.)

For Beck, the question of agency is critical, especially as it might contribute to rhetoricians' experiments in algorithmic and code-based composition. Her questions regarding the impact of nonhuman activity in the construction and dissemination of digital texts are all the more intriguing given the increasing use by developers of script tools, bots (such as Mozilla's patch testing software or Wikipedia's swarm of automated editors), and so on; if we cannot afford to ignore the role of such software agents in our digital composing practices, then how might we re-evaluate what we (and they) are *doing* when we collaboratively compose software?

Ultimately, these examinations of software, code, and code-related discourse as rhetorical and significant forms of meaning making serve

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 to date been underexamined 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.

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 accurately argue 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). But 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, partly because of 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, 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, 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" (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 nonmeaningful

instructions. Admittedly, Burke argued that meaning was limited to human communication (with nonhuman activity instead reflecting nonsymbolic "motion" after Hobbes; see Burke, 1962, 135–37), 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 (2011) 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 constructing meaning (even if not necessarily specific to digital media). When using Shipka's structure, "questions associated with materiality and the delivery, reception, and circulation of texts, objects, and events are less likely to be viewed as separate from or incidental to the means and methods of production, but more likely as integral parts of the invention and production process" (2011, 101). Shipka's framework can apply as easily to code and the technologies that facilitate it as it can 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 (2011) observed, recognizing the equal importance of any and all "modes, materials, methods, and technologies" that may used as part of a given rhetorical activity is integral to their skillful use in moving an audience to engage in some form of action (85). 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 bring about or otherwise influence through the suasive act.

Such a model is aligned with the goals of activity theory (AT) and its computational structure as well. In activity theory, 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[.] (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. 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.

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 construction 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 toward 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 other-

wise 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.

Bibliography

Abbate, J. 2012. *Recoding Gender: Women's Changing Participation in Computing*. Cambridge, MA: MIT Press.

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. 251–55. Retrieved from https://dx.doi.org/10.1109/ICCTD.2009.161

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

Applen, J. D. 2001. "Technical Communication, Knowledge Management, and XML." *Technical Communication* 49 (3): 301–13.

Arns, I. 2005. "Code as Performative Speech Act." *#Artnodes*. Retrieved from http://www.uoc.edu/artnodes/espai/eng/art/arns0505.pdf

Ballentine, B. 2009. "In Defense of Obfuscation: Questioning Open Source and a New Perspective on Teaching Digital Literacy in the Writing Classroom." In *Composition & Copyright: Perspectives on Teaching, Text-Making, and Fair Use*, edited by S. Westbrook, 68–89. Albany: SUNY Press.

Banks, A. 2006. *Race, Rhetoric, and Technology: Searching for Higher Ground*. Mahwah, NJ: Lawrence Erlbaum Press.

Barnett, F., Z. Blas, M. Cardenas, J. Gaboury, J. M. Johnson, and M. Rhee. 2016. "QueerOS: A User's Manual." In *Debates in the Digital Humanities*, edited by M. K. Gold and L. F. Klein. Retrieved from http://dhdebates.gc.cuny.edu/debates/text/56

Bazerman, C. 1994. "Systems of Genres and the Enactment of Social Intentions." In *Genre and the New Rhetoric*, edited by A. Freadman and P. Medway, 79–101. London: Taylor & Francis.

Beck, E. 2016. "A Theory of Persuasive Computer Algorithms for Rhetorical Code Studies." *enculturation* 23. Retrieved from http://enculturation.net/a-theory-of-persuasive-computer-algorithms

Bellinger, M. 2016. "The Rhetoric of Error in Digital Media." *Computational Culture* 5. Retrieved from http://computationalculture.net/the-rhetoric-of-error-in-digital-media-2/

Bénabou, M. 2007. "Rule and Constraint." In *Oulipo: A Primer of Potential Literature*, edited by W. Motte, 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): 359–78.

Berge, C. 2007. "For a Potential Analysis of Combinatory Literature." In *Oulipo: A Primer of Potential Literature*,edited by W. Motte, 115–25. 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

Birkbak, A., and H. B. Carlsen. 2016. "The World of EdgeRank: Rhetorical Justifications of Facebook's News Feed Algorithm." *Computational Culture* 5. Retrieved from http://computationalculture.net/the-world-of-edgerank-rhetorical-justifications-of-facebooks-news-feed-algorithm/

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, M. L. 2015. "A Textual History of Mozilla: Using Topic Modeling to Trace Sociocultural Influences on Software Development." *Digital Humanities Quarterly* 9 (3). Retrieved from http://digitalhumanities.org/dhq/vol/9/3/000224/000224.html

Black, P. E. 2007. "Algorithm." In *Dictionary of Algorithms and Data Structures*, edited by P. E. Black. Retrieved from http://xlinux.nist.gov/dads//HTML/algorithm.html

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. and N. Montfort. 2009. "Platform Studies: Frequently Questioned Answers." *Proceedings of the Digital Arts and Culture Conference*. Retrieved from http://escholarship.org/uc/item/01rok9br

Bossert, T. P. 2017. "It's Official: North Korea Is Behind WannaCry." *Wall Street Journal*. Retrieved from https://www.wsj.com/articles/its-official-north-korea-is-behind-wannacry-1513642537

Bowker, G. 2008. "Software Values." *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Brassard, G., and P. Bratley. 1996. *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice-Hall.

Brock, K. 2014. "Enthymeme as Rhetorical Algorithm." *Present Tense: A Journal of Rhetoric in Society* 4 (1). Retrieved from http://www.presenttensejournal.org/volume-4/enthymeme-as-rhetorical-algorithm/

Brock, K. 2016. "The 'FizzBuzz' Programming Test: A Case-Based Exploration of Rhetorical Style in Code." *Computational Culture: A Journal of Software Studies* 5. Retrieved from http://computationalculture.net/article/the-fizzbuzz-programming-test-a-case-based-exploration-of-rhetorical-style-in-code

Brock, K., and A. R. Mehlenbacher. 2017. "Rhetorical Genres in Code." *Journal of Technical Writing and Communication*. http://dx.doi.org/10.1177/0047281617726278

Brock, K., and D. Shepherd. 2016. "Understanding How Algorithms Work Persuasively through the Procedural Enthymeme." *Computers and Composition* 42: 17–27.

Brooke, C. G. 2009. *Lingua Fracta: Towards a Thetoric of New Media*. Cresskill, NJ: Hampton Press.

Brown, J. J., Jr. 2015. *Ethical Programs: Hospitality and the Rhetorics of Software*. Ann Arbor: University of Michigan Press.

Brown, J. J., Jr., and A. Vee. 2016. "Rhetoric Special Issue Editorial Introduc-

tion." *Computational Culture* 5. Retrieved from http://computationalculture.net/rhetoric-special-issue-editorial-introduction/

Burgess, H. 2010. "<?php>: 'Invisible' Code and the Mystique of Web Writing." In *From a to <a>: Keywords of Markup*, edited by B. Dilger & J. Rice, 167–85. Minneapolis: University of Minnesota Press.

Burke, K. 1962. *A Grammar of Motives*. Berkeley: University of California Press.

Burke, K. 1969. *A Rhetoric of Motives*. Berkeley: University of California Press.

Burke, K. 1973. *The Philosophy of Literary Form*. Berkeley: University of California Press.

Burleigh, N. 2015. "What Silicon Valley Thinks of Women." *Newsweek*. Retrieved from http://www.newsweek.com/2015/02/06/what-silicon-valley-thinks-women-302821.html

Carnegie, T. A. M. 2009. "Interface as Exordium: The Rhetoric of Interactivity." *Computers and Composition* 26 (3): 164–73.

Carpenter, R. 2009. "Boundary Negotiations: Electronic Environments as Interface." *Computers and Composition* 26 (3): 138–48.

Carroll, L. 1973. *Symbolic Logic*. Edited by W. W. Bartley. New York: Clarkson N. Potter.

Cassidy, S. 2014. "Diagnosis of the OpenSSL Heartbleed Bug." *Sean Cassidy*. Retrieved from https://www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html

Castiglione, B. 1988. *The Book of the Courtier*. Translated by George Bull. New York: Penguin.

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: University of Chicago Press.

Christiansen, E. 1996. "Tamed By a Rose: Computers as Tools in Human Activity." In *Context and Consciousness: Activity Theory and Human-Computer Interaction*, edited by B. Nardi, 175–98. Cambridge, MA: MIT Press.

Christley, S., and G. Madey. 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 https://dx.doi.org/10.1109/HICSS.2007.74

Chun, W. H. K. 2011. *Programmed Visions: Software and Memory*. Cambridge, MA: MIT Press.

Coleman, E. G. 2013. *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton: Princeton University Press.

Cooper, M. M. 1986. "The Ecology of Writing." *College English* 48 (4): 364–75.

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

Crowley, S., and D. Hawhee. 2009. *Ancient Rhetorics for Contemporary Students*. 4th ed. New York: Pearson.

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)*. Genoa, Italy. Retrieved from http://floss.syr.edu

Crowston, K., and J. Howison. 2005. "The Social Structure of Free and Open Source Software Development." *First Monday* 10: 1–27. Retrieved from http://floss.syr.edu

Crowston, K., and J. Howison. 2006. "Assessing the Health of Open Source Communities." *IEEE Computer* 39: 113–15. 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): 430–43.

deasydoesit, et al. 2018. "Object Prototypes." *MDN Web Docs*. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Debian. 2011. "Debian Project Leader." *Debian Developer's Corner*. Retrieved from http://www.debian.org/devel/leader.en.html

Divine, D., T. Ferro, and M. Zachry. 2011. "Work Through the Web: A Typology of Web 2.0 Services." In *SIGDOC '11: Proceedings of the 29th Annual ACM International Conference on Design of Communication*, edited by A. Protopsaltis et al., 121–28. New York: ACM Digital Library.

dmose%netscape.com, and peterv. 2002. "Initial work (mostly by peterv) for a bayesian spam filter. Not yet part of the build." *GitHub*. Retrieved from https://github.com/mozilla/ gecko-dev/blob/b7890a4bbb26100ef95f8872b10e18349d961400/mailnews/extensions/bayesian-spam-filter/menuOverlay.js

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

Dubinsky, J. M., ed. 2004. *Teaching Technical Communication: Critical Issues for the Classroom*. Boston: Bedford/St. Martin's Press.

Duckett, J. 2014. *JavaScript and jQuery: Interactive Front-End Web Development*. Indianapolis: Wiley & Sons.

Dyehouse, J. 2007. "Knowledge Consolidation Analysis: Toward a Methodology for Studying the Role of Argument in Technology Development." *Written Communication* 24 (2): 111–39.

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 University Press.

ehsan, et al. 2013. "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/gecko-dev/blob/88ce997385044499f1781dcf7b80a1a969e282d8/browser/base/content/browser.js

Eyman, D. 2015. *Digital Rhetoric: Theory, Method, Practice*. Ann Arbor: University of Michigan Press.

Fagerjord, A. 2003. "Rhetorical Convergence: Studying Web Media." In *Digital Media Revisited*, edited by G. Liestøl, A. Morrison, and T. Rasmussen, 293–325. Cambridge, MA: MIT Press.

Fahnestock, J. 2011. *Rhetorical Style: The Uses of Language in Persuasion*. Oxford: Oxford University Press.

Ford, P. 2005. "Processing Processing." In *The Best Software Writing I*, edited by J. Spolsky, 79–94. Berkeley, CA: Apress.

Ford, P. 2015. "What Is Code?" *Businessweek*. Retrieved June 11, 2015, from http://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/

França, R., 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, NY: Autonomedia.

Fuller, M., and S. Matos. 2011. "Feral Computing: From Ubiquitous Calculation to Wild Interactions." *The Fibreculture Journal*, 144–63. 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 *Rhetorical Hermeneutics: Invention and Interpretation in the Age of Science*, edited by A. G. Gross and W. M. Keith, 279–98. Albany: State University of New York Press.

Gallagher, J. R. 2017. "Writing for Algorithmic Audiences." *Computers and Composition* 45: 25–35.

Gaonkar, D. P. 1997. "The Idea of Rhetoric in the Rhetoric of Science." In *Rhetorical Hermeneutics: Invention and Interpretation in the Age of Science*, edited by A. G. Gross and W. M. Keith, 25–85. Albany: SUNY Press.

Gaonkar, D. P. 2004. "Introduction: Contingency and Probability." In *A Companion to Rhetoric and Rhetorical Criticism*, edited by W. Jost and W. Olmsted, 6–21. Oxford, UK: Blackwell Publishing.

Garsten, B. 2006. *Saving Persuasion: A Defense of Rhetoric and Judgment*. Cambridge, MA: Harvard University Press.

gbrownmozilla. 2011. "Bug 669549—Some DeviceManager ADB functions do not work; r=jmaher a=test-only." *GitHub*. Retrieved from https://github.com/mozilla/gecko-dev/commit/6ccc1a61a0bd87fa5144aa427d697c1971b1cacd

Gerrard, L. 1995. "The Evolution of the Computers and Writing conference." *Computers and Composition* 12 (3): 279–92.

Gillespie, T. 2014. "The Relevance of Algorithms." In *Media Technologies: Essays on Communication, Materiality, and Society*, edited by T. Gillespie, P. J. Boczkowski, and K. A. Foot, 167–94. Cambridge, MA: MIT Press.

Gillespie, T. 2016. Algorithm. In *Digital Keywords: A Vocabulary of Information Society & Culture*, edited by B. Peters, 18–30. Princeton: Princeton University Press.

GIMP. 2018. "Getting Involved." Retrieved from http://www.gimp.org/develop

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

GitHub. 2017. *Open Source Survey*. Retrieved from http://opensourcesurvey.org/2017/

Goodling, L. 2015. "MOAR Digital Activism, Please." *Kairos: A Journal of Rhetoric, Technology, and Pedagogy* 19 (3). Retrieved from http://kairos.technorhetoric.net/19.3/topoi/goodling/index.html

Gozala 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): 455–72.

Gurak, L. 1997. *Persuasion and Privacy in Cyberspace*. New Haven: Yale University Press.

Haefner, J. 1999. "The Politics of the Code." *Computers and Composition* 16 (3): 325–39.

Hairston, M. C. 1986. "Bringing Aristotle's Enthymeme into the Composition Class-

room." In *Rhetoric and Praxis: The Contribution of Classical Rhetoric to Practical Reasoning*, edited by J. D. Moss, 59–77. Washington, DC: Catholic University of America Press.

Hamerly, J., T. Paquin, and S. Walton. 1999. "Freeing the Source: The Story of Mozilla." In *Open Sources: Voices from the Revolution*, edited by C. DiBona and S. Ockman. Retrieved April 17, 2012, from http://oreilly.com/openbook/opensources/book/netrev.html

Hart-Davidson, W., et al. 2008. "Coming to Content Management: Inventing Infrastructure for Organizational Knowledge Work." *Technical Communication Quarterly* 17 (1): 10–34.

Haverbeke, M. 2015. *Eloquent JavaScript: A Modern Introduction to Programming*. 2nd ed. San Francisco: No Starch Press.

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: University of Chicago Press.

Hayles, N. K. 2012. *How We Think: Digital Media and Contemporary Technogenesis*. Chicago: University of Chicago Press.

headius, et al. 2012. "Incorporate OpenSSL tests from JRuby." *GitHub*. Retrieved from https://github.com/ruby/ruby/pull/206

Helmond, A. 2013. "The Algorithmization of Hyperlinks." *Computational Culture* 3. Retrieved from http://computationalculture.net/the-algorithmization-of-the-hyperlink/

Herrick, J. A. 2016. *History and Theory of Rhetoric: An Introduction*. 5th ed. New York: Routledge.

Hicks, M. 2017. *Programmed Inequality: How Britain Discarded Women Technologists and Lost its Edge in Computing*. Cambridge, MA: MIT 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): 629–56.

Hodgson, J. and Barnett, S. 2016. "Introduction: What Is Rhetorical about Digital Rhetoric? Perspectives and Definitions of Digital Rhetoric." *enculturation* 23. Retrieved from http://enculturation.net/what-is-rhetorical-about-digital-rhetoric

Holmes, S. 2016. "Ethos, Hexis, and the Case for Persuasive Technology." *enculturation* 23. Retrieved from http://enculturation.net/ethos-hexis-and-the-case-for-persuasive-technologies

Holmes, S. 2017. *The Rhetoric of Videogames as Embodied Practice: Procedural Habits*. New York: Routledge.

Howison, J. 2006. *Coordinating and Motivating Open Source Contributors*. Retrieved from http://floss.syr.edu

Howison, J., K. Inoue, and K. Crowston. 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)*. Translated by S. Torvalds. Retrieved from http://openlife.cc/onlinebook

Ingraham, C. 2014. "Toward an Algorithmic Rhetoric." In *Digital Rhetoric and Global Literacies*, edited by G. Verhulsdonck and M. Limbu, 62–79. Hershey, PA: IGI Global.

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, J. M., and M. A. Neal. 2017. "Introduction: Wild Seed in the Machine." *Black Scholar: Journal of Black Studies and Research* 47 (3). https://doi.org/10.1080/00064246.2017.1329608

Johnson, N. R. 2014. "Protocological Rhetoric: Intervening in Institutions." *Journal of Technical Writing and Communication* 44 (4): 381–400.

Johnson, R. R. 1998. "Complicating Technology: Interdisciplinary Method, the Burden of Comprehension, and the Ethical Space of the Technical Communicator." *Technical Communication Quarterly* 7 (1): 75–98.

Johnson, R. R. 1999. "Johnson Responds." *Technical Communication Quarterly* 8 (2): 224–26.

Juszkiewicz, J., and J. Warfel. 2016. "The Rhetoric of Mathematical Programming." *enculturation* 23. Retrieved from http://enculturation.net/the-rhetoric-of-mathematical-programming

Keith, W. M. 1997. "Engineering Rhetoric." In *Rhetorical Hermeneutics: Invention and Interpretation in the Age of Science*, edited by A. G. Gross and W. M. Keith, 225–46. Albany: SUNY Press.

Kelty, C. M. 2008. *Two Bits: The Cultural Significance of Free Software*. Durham, NC: Duke University Press.

Kerner, S. M. 2015. "Why All Linux (Security) Bugs Aren't Shallow." *eSecurity Planet*. Retrieved from https://www.esecurityplanet.com/open-source-security/why-all-linux-security-bugs-arent-shallow.html

Kernighan, B. W., and R. Pike. 1999. *The Practice of Programming*. Reading, MA: Addison-Wesley.

Kernighan, B. W., and P. J. Plauger. 1978. *The Elements of Programming Style*. 2nd ed. New York: McGraw-Hill.

Khomami, N., and O. Solon. 2017. "'Accidental Hero' Halts Ransomware Attack and Warns: This Is Not Over." *The Guardian*. Retrieved from https://www.theguardian.com/technology/2017/may/13/accidental-hero-finds-kill-switch-to-stop-spread-of-ransomware-cyber-attack

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

Kimme Hea, A. C. 2007. "Riding the Wave: Articulating a Critical Methodology for Web Research Practices." In *Digital Writing Research: Technologies, Methodologies, and Ethical Issues*, edited by H. A. McKee and D. N. DeVoss, 269–86. Cresskill, NJ: Hampton Press.

Kitchin, R., and M. Dodge. 2011. *Code/Space: Software and Everyday Life*. Cambridge, MA: MIT Press.

Kittler, F. 2008. "Code." In *Software Studies: A Lexicon*, edited by M. Fuller. Translated by T. Morrison and F. Cramer, 40–47. Cambridge, MA: MIT Press.

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

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/gecko-dev/blob/5d30f398bd39d63e9938165f9def84e2218c8589/browser/base/content/browser.js

Lanham, R. A. 1993. *The Electronic Word: Democracy, Technology, and the Arts*. Chicago: University of Chicago Press.

Lanham, R. A. 2003. *Analyzing Prose*. 2nd ed. New York: Continuum.

Le Lionnais, F. 2007. "Lipo: First Manifesto." In *Oulipo: A Primer of Potential Literature*, edited by W. Motte, 26–28. Champaign, IL: Dalkey Archive Press.

Lee, M., W. Mercer, P. Rascagneres, and C. Williams. 2017. "Player 3 has entered the game: Say hello to 'WannaCry.'" *Talos*. Retrieved from http://blog.talosintelligence.com/2017/05/wannacry.html

Leyden, J. 2014. "AVG on Heartbleed: It's Dangerous to Go Alone, Take This (an AVG Tool). *The Register*. Retrieved from https://www.theregister.co.uk/2014/05/20/heartbleed_still_prevalent/

Leyden, J. 2017. "WannaCry Ransomware Note Likely Written by Google Translate – Using Chinese Speakers." *The Register*. Retrieved from https://www.theregister.co.uk/2017/05/26/wannacrypt_ransom_note_linguistics/

Lopes, C. V. 2014. *Exercises in Programming Style*. Boca Raton, FL: CRC Press.

Lopez, G. 2017. "Diversity Problems in the Tech Industry Go Far Beyond Google." *Vox*. Retrieved from https://www.vox.com/identities/2017/8/8/16113070/google-memo-diversity-tech

Losh, E. 2009. *Virtualpolitik: An Electronic History of Government Media-Making in a Time of War, Scandal, Disaster, Miscommunication, and Mistakes*. Cambridge, MA: MIT Press.

Losh, E. 2016. "Sensing Exigence: A Rhetoric for Smart Objects." *Computational Culture* 5. Retrieved from http://computationalculture.net/sensing-exigence-a-rhetoric-for-smart-objects/

Lovelace, A. 2002. "Sketch of the Analytical Engine." In *Literature and Science in the Nineteenth Century: An Anthology*, edited by L. Otis, 15–19. Oxford: Oxford University 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: Princeton University Press.

Maher, J. 2011. "The Technical Communicator as Evangelist: Toward Critical and Rhetorical Literacies of Software Documentation." *Journal of Technical Writing and Communication* 41 (4): 367–401.

Maher, J. 2016. "Artificial Rhetorical Agents and the Computing of Phronesis." *Computational Culture* 5. Retrieved from http://computationalculture.net/artificial-rhetorical-agents-and-the-computing-of-phronesis/

Manovich, L. 2001. *The Language of New Media*. Cambridge, MA: MIT Press.

Manovich, L. 2008. *Software Takes Command*. Retrieved from http://www.softwar estudies.com/softbook/manovich_softbook_11_20_2008.pdf

Marino, M. 2006. "Critical Code Studies." *Electronic Book Review*. Retrieved from http://www.electronicbookreview.com/thread/electropoetics/codology

Mason, P. 2016. "The Racist Hijacking of Microsoft's Chatbot Shows How the Internet Teems with Hate." *The Guardian*. Retrieved from https://www.theguardian.com/world/2016/mar/29/microsoft-tay-tweets-antisemitic-racism

Mateas, M. 2005. "Procedural Literacy: Educating the New Media Practitioner." *On the Horizon* 13 (2): 101–11.

Matsumoto, Y. 2007. "Treating Code As an Essay." In *Beautiful Code: Leading Programmers Explain How They Think*, edited by A. Oram and G. Wilson, 477–81. Sebastopol, CA: O'Reilly.

McCorkle, B. 2012. *Rhetorical Delivery as Technological Discourse*. Carbondale: Southern Illinois University Press.

McNenny, G., and D. Roen. 1992. "The Case for Collaborative Scholarship in Rhetoric and Composition." *Rhetoric Review* 10 (2): 291–310.

McPherson, T. 2012. "Why Are the Digital Humanities So White? Or Thinking the Histories of Race and Computation." In *Debates in the Digital Humanities*, edited by M. K. Gold, 139–60. Minneapolis: University of Minnesota Press.

Miller, C. R. 1979. "A Humanistic Rationale for Technical Writing." *College English* 40 (6): 610–17.

Miller, C. R. 1984. "Genre as Social Action." *Quarterly Journal of Speech* 70 (2): 151–67.

Miller, C. R. 2000. "The Aristotelian *Topos*: Hunting for Novelty." In *Rereading Aristotle's* Rhetoric, edited by A. Gross and A. Walzer. Carbondale: Southern Illinois University Press.

Miller, C. R. 2001. "Writing in a Culture of Simulation: Ethos Online." In *The Semiotics of Writing: Transdisciplinary Perspectives on the Technology of Writing*, edited by P. Coppock, 253–79. Turnhout, Belgium: Brepols Publishers.

Miller, C. R. 2007. "What Can Automation Tell Us about Agency?" *Rhetoric Society Quarterly* 37 (2): 137–57.

Miller, C. R. 2010a. "Foreword: Rhetoric, Technology, and the Pushmi-Pullyu." In *Rhetorics and Technologies: New Directions in Writing and Communication*, edited by S. A. Selber, ix–xii. Columbia: University of South Carolina Press.

Miller, C. R. 2010b. "Should We Name the Tools? Concealing and Revealing the Art of Rhetoric." In *The Public Work of Rhetoric: Citizen-Scholars and Civic Engagement*, edited by J. M. Ackerman and D. J. Coogan, 19–38. Columbia: University of South Carolina Press.

Miller, C. R., and D. Shepherd. 2004. "Blogging as Social Action: A Genre Analysis of the Weblog." In *Into the Blogosphere: Rhetoric, Community, and Culture of Weblogs*, edited by L. Gurak and S. Antonijevic. University of Minnesota. Retrieved from https://conservancy.umn.edu/handle/11299/172818

Miller, C. R., and D. Shepherd. 2009. "Questions for Genre Theory from the Blogosphere." In *Genres in the Internet: Issues in the Theory of Genre*, edited by J. Giltrow and D. Stein, 263–90. Amsterdam: John Benjamins.

Montfort, N., and I. Bogost. 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

Moore, P. 1999. "Myths about Instrumental Discourse: A Response to Robert R. Johnson." *Technical Communication Quarterly* 8 (2): 210–23.

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. 2016. "EngineeringProductivity/Projects/Treeherder." *Mozilla Wiki*. Retrieved from https://wiki.mozilla.org/EngineeringProductivity/Projects/Treeherder

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., K. Yamada, and E. Giaccardi. 2005. "Understanding the Nature of Collaboration in Open-Source Software Development." Proceedings of the 12th Asia-Pacific software engineering conference. Taipei, Taiwan. Retrieved from https://doi.org/10.1109/APSEC.2005.108

Nakamura, L. 2007. *Digitizing Race: Visual Cultures of the Internet*. Minneapolis: University of Minnesota Press.

Nardi, B. A. 1995. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press.

Nicotra, J. 2016. "Assemblage Rhetorics: Creating New Frameworks for Rhetorical Action." In *Rhetoric, through Everyday Things*, edited by S. Barnett and C. Boyle, 185–96. Tuscaloosa: University of Alabama Press.

Noble, S. U. 2018. *Algorithms of Oppression: How Search Engines Reinforce Racism*. New York: New York University Press.

Nowviskie, B. 2014. "Ludic algorithms." In *Pastplay: Teaching and Learning History with Technology*, edited by K. Kee, 139–71. Ann Arbor: University of Michigan Press.

Ong, W. J. 2002. *Orality and Literacy*. New York: Routledge.

O'Reilly, T. 2005. "The Open Source Paradigm Shift." In *Open Sources 2.0: The Continuing Evolution*, edited by C. DiBona, D. Cooper, and M. Stone, 253–72. Sebastopol, CA: O'Reilly Media.

Parikka, J. 2008. "Copy." In *Software Studies | A Lexicon*, edited by M. Fuller. Cambridge, MA: MIT 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): 207–24.

Prelli, L. J. 1989. "The Rhetorical Construction of Scientific Ethos." In *Rhetoric in the*

Human Sciences: Inquiries in Social Construction, edited by H. W. Simmons, 48–68. London: Sage.

Queneau, R. 2007. "Potential Literature." In *Oulipo: A Primer of Potential Literature*, edited by W. Motte, 51–64. Champaign, IL: Dalkey Archive Press.

Queneau, R. 2009. *Exercises in Style*. Translated by B. Wright. New York: New Directions.

"Quine." n.d. In *Rosetta Code*. Retrieved from http://rosettacode.org/wiki/Quine

Quine, W. V. 1976. "The Ways of Paradox." In *The Ways of Paradox and Other Essays*, W. V. Quine. Cambridge, MA: Harvard University Press. Retrieved from http://www.math.dartmouth.edu/~matc/Readers/HowManyAngels/Paradox.html

Rahm, E., et al. 2016. "Bug 1286041—1,700 instances of 'data callback fires before cubeb_stream_start() is called' emitted from dom/media/AudioStream.cpp during linux64 debug testing." *Bugzilla@Mozilla*. Retrieved from https://bugzilla.mozilla.org/show_bug.cgi?id=1286041

Ramsay, S. 2011. *Reading Machines: Toward an Algorithmic Criticism*. Urbana: University 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

Reagle, J. 2013. "'Free as in sexist?': Free Culture and the Gender Gap." *First Monday* 18 (1). Retrieved from http://journals.uic.edu/ojs/index.php/fm/article/view/4291/3381

Red Hat. 2017. "Fedora's Mission and Foundations." *Fedora*. Retrieved from https://docs.fedoraproject.org/fedora-project/project/fedora-overview.html

Reif, A. 2013. "Feminism and Biases." *HASTAC*. Retrieved from https://www.hastac.org/comment/8399#comment-8399

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/

Rieder, D. 2016. "Making wayves." *enculturation* 23. Retrieved from http://enculturation.net/making-wayves

Risam, R. 2015. "Beyond the Margins: Intersectionality and the Digital Humanities." *Digital Humanities Quarterly* 9 (2). Retrieved from http://www.digitalhumanities.org/dhq/vol/9/2/000208/000208.html

Ritzer, G., and N. Jurgenson. 2010. "Production, Consumption, Prosumption: The Nature of Capitalism in the Age of the Digital 'Prosumer.'" *Journal of Consumer Culture* 10 (1): 13–36. https://doi.org/10.1177/1469540509354673

Rose, A. 2010. Are Face-Detection Cameras Racist? *Time*. Retrieved from http://content.time.com/time/business/article/0,8599,1954643,00.html

Rushkoff, D. 2011. *Program or Be Programmed: Ten Commands for a Digital Age*. Berkeley, CA: Soft Skull Press.

Sack, W. 2008. "From Software Studies to Software Design." *SoftWhere 2008 Software Studies Workshop*. Retrieved from http://workshop.softwarestudies.com

Salter, A., and J. Murray. 2014. *Flash: Building the Interactive Web*. Cambridge, MA: MIT Press.

Sample, M. 2013. "Criminal Code: Procedural Logic and Rhetorical Excess in Videogames." *Digital Humanities Quarterly* 7 (1). Retrieved from http://digitalhumanities.org/dhq/vol/7/1/000153/000153.html

saskatchewancatch, et al. 2016. "Introduction to Object-Oriented JavaScript." *Mozilla Developer Network*. Retrieved from https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript

Schlesinger, A. 2013. "Feminism and Programming Languages." *HASTAC*. Retrieved from https://www.hastac.org/blogs/ari-schlesinger/2013/11/26/feminism-and-programming-languages

Schmidt, B. M. 2016. "Do Digital Humanists Need to Understand Algorithms?" In *Debates in the Digital Humanities*, edited by M. K. Gold and L. F. Klein. Retrieved from http://dhdebates.gc.cuny.edu/debates/text/99

Scott, Z., et al. 2012. "Feature #7400: Incorporate OpenSSL Tests from JRuby." *Ruby Issue Tracking System*. Retrieved from http://bugs.ruby-lang.org/issues/7400

Selber, S. A. 2004. *Multiliteracies for a Digital Age*. Carbondale: Southern Illinois University Press.

Selfe, C. L., and R. J. Selfe 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/gecko-dev/blob/443b32151a08foad62d39291b4ad5e7d5b65521e/browser/components/preferences/main.js

Shepherd, D. 2016. *Building Relationships: Online Dating and the New Logics of Internet Culture*. London: Lexington Books.

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. 2014. "HashMapClass." *Processing*. Retrieved from https://github.com/processing/processing-docs/tree/master/content/examples/Topics/Advanced%20Data/HashMapClass

Shipka, J. 2011. *Toward a Composition Made Whole*. Pittsburgh: University of Pittsburgh Press.

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): 187–99.

Smith, V. J. 2007. "Aristotle's Classical Enthymeme and the Visual Argumentation of the Twenty-First Century." *Argumentation and Advocacy* 43 (3–4): 114–23. https://doi.org/10.1080/00028533.2007.11821667

snhenson, et al. 2015. "Add heartbeat extension bounds check." *GitHub*. Retrieved from https://github.com/openssl/openssl/commit/731f431497f463f3a2a97236fe0187b11c44aead

Spinuzzi, C. 2002a. "Software Development as Mediated Activity: Applying Three Analytical Frameworks for Studying Compound Mediation." In *SIGDOC '01: Proceedings of the 19th Annual International Conference on Computer Documentation*, edited by M. J. Northrop and S. Tilley, 58–67. New York: ACM Digital Library.

Spinuzzi, C. 2002b. "Towards a Hermeneutic Understanding of Programming Languages." *Currents in Electronic Literacy* 6. Retrieved from http://currents.dwrl.utexas.edu/spring02/spinuzzi.html

Spinuzzi, C. 2003. "Compound Mediation in Software Development." In *Writing Selves/Writing Societies: Research from Activity Perspectives*, edited by C. Bazerman and D. R. Russell. Fort Collins, CO: The WAC Clearinghouse.

Steinberg, J. 2014. "Sexism in Startups: The Frank Conversation We Need to Be Having." *Forbes*. Retrieved from https://www.forbes.com/sites/josephsteinberg/2014/09/18/heres-the-conversation-men-need-to-have-about-sexism-as-told-by-a-man/

Steiner, C. 2012. *Automate This: How Algorithms Came to Rule Our World*. New York: Penguin.

Stewart, D. 2005. "Social Status in an Open-Source Community." *American Sociological Review* 70 (5): 823–42.

Stolley, K. 2014. "MVC, Materiality, and the Magus: The Rhetoric of Source-Level Production." In *Rhetoric and the Digital Humanities*, edited by J. Ridolfo and W. Hart-Davidson, 264–76. Chicago: University of Chicago Press.

Streuver, N. S. 2009. *Rhetoric, Modality, Modernity*. Chicago: University of Chicago Press.

Sullivan, R. 2013. "The Liminal Textuality of Comments in Code." Presented at the Objects of Textual Scholarship, Society for Textual Scholarship (STS) Conference. Chicago. Retrieved from http://rachaelsullivan.com/sts2013/index.html

Swarts, J. 2011. "Technological Literacy as Network Building." *Technical Communication Quarterly* 20 (3): 274–302.

Swarts, J. 2015. "Help Is in the Helping: An Evaluation of Help Documentation in a Networked Age." *Technical Communication Quarterly* 24 (2): 164–87.

Tarsa, R. 2015. "Upvoting the Exordium: Literacy Practices of the Digital Interface." *College English* 78 (1): 12–33.

Thomson, P. 2008. "Git vs. Mercurial: Please Relax." *Important Shock*. Retrieved from https://importantshock.wordpress.com/2008/08/07/git-vs-mercurial/

Thompson, K. 1984. "Reflections on Trusting Trust." *Communications of the ACM* 27 (8): 761–63.

topwiz. 2013. "The Real World." *HASTAC*. Retrieved from https://www.hastac.org/comment/8380#comment-8380

troy%netscape.com. 1999. Work-in-progress for 'min' and 'max' properties. *GitHub*. Retrieved from https://github.com/mozilla/gecko-dev/commit/a6281248abc3cb214706f970205b63ae5c83c832

Truscello, M. 2005. "The Rhetorical Ecology of the Technical Effect." *Technical Communication Quarterly* 14 (3): 345–51.

Valarissa, et al. 2012. "Pull request #554: Fixed a small, but major, typo." *GitHub*. Retrieved from https://github.com/hotsh/rstat.us/pull/554

Vatz, R. E. 1968. "The Myth of the Rhetorical Situation." *Philosophy & Rhetoric* 6 (3): 154–61.

Vee, A. 2017. *Coding Literacy: How Computer Programming Is Changing Writing*. Cambridge, MA: MIT Press.

Walden, J., B. Goodger, and A. Romano. 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/gecko-dev/commit/d50971d58f8d4c521a490c1e18c3e4b2a4e9ad07#diff-37f10ac89685d3613a8073b8f663becb

Walden, J., B. Goodger, and A. Romano. 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/gecko-dev/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.

Walton, D. 2001. "Enthymemes, Common Knowledge, and Plausible Inference." *Philosophy and Rhetoric* 34 (2): 93–112.

Wardrip-Fruin, N. 2009. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. Cambridge, MA: MIT Press.

Warnick, B. 2004. "Online Ethos: Source Credibility in an 'Authorless' Environment." *American Behavioral Scientist* 48 (2): 256–65.

Warnick, B. 2007. *Rhetoric Online: Persuasion and Politics on the World Wide Web*. New York: Peter Lang.

Warnock, S., and M. Kahn. 2007. "Expressive/Exploratory Technical Writing (XTW) in Engineering: Shifting the Technical Writing Curriculum." *Journal of Technical Writing and Communication* 37 (1): 37–57.

Wiggins, A., J. Howison, and K. Crowston. 2008. "Social Dynamics of FLOSS Team Communication across Channels." Proceedings of the IFIP 2.13 Working Conference on Open Source Software (OSS). Milan, Italy. 131–42. Retrieved from http://floss.syr.edu

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

W3Counter. 2017. "Browser & Platform Market Share: December 2017." *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): 319–25.

Zawinski, J. 2004. "Censorzilla." *JWZ.org*. Retrieved from https://www.jwz.org/doc/censorzilla.html

Index