

Rhetorical Genres in Code

Kevin Brock¹ and
Ashley Rose Mehlenbacher²

Journal of Technical Writing and
Communication
0(0) 1–29

© The Author(s) 2017

Reprints and permissions:

sagepub.com/journalsPermissions.nav

DOI: 10.1177/0047281617726278

journals.sagepub.com/home/jtw



Abstract

We examine the rhetorical activity employed within software development communities in code texts. For technical communicators, the rhetoricity of code is crucial for the development of more effective code and documentation. When we understand that code is a collection of rhetorical decisions about how to engage those machinic processes, we can better attend to the significance and nuance of those decisions and their impact on potential user activities.

Keywords

code studies, software studies, digital rhetoric, rhetorical genre studies

Introduction

Code serves as textual and procedural description of, as well as medium for, the construction of software, but in the overwhelming majority of cases—even when a given program might impact millions of people—code is infrequently examined as a form of meaningful communication. Practicing technical communicators have long understood the rhetorical apparatus of code in regard to related texts and contexts, for example, documentation and code comments (lines of text read by humans but explicitly ignored by machines). Comments can help more clearly explain the way a given body of code was written and how it functions

¹University of South Carolina, Columbia, SC, USA

²University of Waterloo, Ontario, Canada

Corresponding Author:

Kevin Brock, University of South Carolina, Humanities Office Building, Columbia, SC 29208, USA.

Email: brockkm2@mailbox.sc.edu

(or occasionally does not, despite the motives and intentions of its authors). Comments often include notes about “best practices,” as well as the “elegance” and “efficiency” of code, either specifically or in general—often in comparison of a given block of code with an assumed, subjective ideal. Investigating code comments shows us that conventions and values of a development community are constantly instantiated, challenged, and negotiated via a variety of rhetorical strategies. Although comments are sites of important rhetorical work in code, we argue that important contributions examining the rhetorical function and capacity of code itself can be made.

Others have noted the important insights rhetorical study reveals about the nature of software, such as Brown and Vee’s (2016) special issue of *Computational Culture* on rhetoric and computation, which is filled with articles exploring the implications of identifying and examining code and software as forms of, and not merely instrumental vehicles for, rhetorical communication. For example, Birkbak and Carlsen’s (2016) study of Facebook’s news feed algorithm and the rhetorical justifications made for and about its development and use in “explicit attempts at envisioning the just public” (n.p.). In another example from the special issue, Holmes (2016) examines excerpts from the code of the computer game *FreeCiv*, as well as the context surrounding its development and use, in order to investigate the extent to which code does and does not reveal “its Being [sic] in representation and use” (n.p.). For technical communicators, an acknowledgment that code *can* and *should* be examined as rhetorical is significant because we can not only develop more effective code documentation but also facilitate a documentation process where technical communicators are not merely scribes for a postproduction packaging of software, but rather where technical communicators are central to the design and development of software and its documentation reflects the epistemological and axiological values shaping code.

In this article, we join the current conversations about the rhetorical nature of computational systems, namely code. With this research, we wish to investigate how code might produce and reproduce certain norms, values, and social actions that impact not only developers but also the users whose experiences are influenced by developers’ decisions about how a program will or should operate. We work from the initial premise that code, if investigated *as a form of written communication*, should possess features of meaningful communicative effort similar to and entirely distinct from those of more traditional discursive writing. One way we understand code as similar to more traditional forms of communication is that it indeed is laden with norms and values, and further, that these norms and values shape the form of communications and their outcomes. Further, we suggest that what we would typically call rhetorical activity—argument, deliberation, per-, and dissuasion—are at work in the process of developing code, just as such activity occur in other means of communicating and facilitating action (Spinuzzi, 2002). More specifically, we argue that in such

a constrained medial form, there is a significant amount of typification in the forms of communication, and so we deploy the tools of genre studies as one means of exploring how rhetoric may operate in code. Specifically, to understand how norms and values may be produced and reproduced in code, we study how genre-like typifications may elucidate conventions and practices that shape the specific ways code functions. Understanding what, if any, rhetorical life code has can help us prepare our students to participate in code development cycles in an effort to enhance technical communications rather than taking on more traditional roles that package content for end users. Motivating this work is the idea that when we as scholars and practitioners of technical communication understand that code is not just a set of instrumental, machinic processes but also a collection of complex design (perhaps rhetorically shaped) decisions about how to engage machinic processes and subsequent uses thereof, we can attend to—as critics, educators, contributors, and practitioners—the significance, impact, and nuance of those decisions.

To test the premises of our argument, we explore, via scrutiny of a selection of related code texts and the contexts surrounding their development, what rhetorical activity appears within coding communities as means of creating social action by which software developers persuade one another to code toward particular ends (instrumental and rhetorical), influenced and informed by their relation to one or more given communities and how their code contributions may affect those relationships. Social actions may include identifying, organizing, and negotiating community membership and hierarchies, which allow particular individuals to code and not others. Relying on a familiar framework to those in professional and technical communication scholarship, we deploy the tools of rhetorical genre studies to investigate these coding communities and their code. Although code may at first seem to reside in venues that lie outside the bounds of many of our everyday inquiries as professional communicators, code and coding activities influence more than the products or processes communicators write *about* but also the processes they write *with* (XML, DITA, HTML5, JavaScript, etc.) and thus influence the content of a technical communicators work. Indeed, as technical communication evolves as a profession some have argued technical communicators are particularly well poised to further engage the design and development of software systems (Albers, 2009; Lewis, 2016; Sánchez, 2016; Sapienza, 2002), including open source systems (Johnson-Eilola, 2002).

The purpose of our project is to explore how technical communicators better understand how computer code enacts certain values and norms (e.g., institutional aims, community social hierarchies, individual preferences in naming styles), which are traceable through code—in its logic, structures, naming patterns, means of dissemination, and modification. Such an explication of the rhetoricity of code is useful because it helps technical communicators might better integrate within code development processes. By integrating within

development processes, we mean become actively engaged as designers by identifying explicitly the kinds of decision-making in response to some rhetorical situation (i.e., the exigence to develop the code). Exploring this line of argument, our thought experiment was that we might find evidence of rhetorical activities by looking for features usually associated with “genres” of composition (namely, typification of function, form, and performance). Discovering evidence of genre-ing activity in code may tell us something about the potentially rhetorical nature of code in a way that existing arguments for code-as-rhetoric have not illuminated. As a result, we grounded our thought experiment in lessons from digital rhetorics and code studies. Code, these areas of scholarship would tell us, indeed has a *rhetoricity*—and we took this as a point to begin investigating if and how we might discover such evidence. It was—and still is—certainly possible that genre is not the best tool for such a job; we determined that, even if we could not find evidence of genre-ing activity, we believed this initial exploration might tell us something interesting about the rhetorical (or arhetorical, perhaps) world of artificial languages.

All told, our goal is to better understand how computer code is composed through a rhetorically driven analysis of coding activities. To fulfill our purpose and achieve our goals, the following question guided our rhetorical analysis: If we accept the idea that code is rhetorical, what might we learn about its rhetorical nature by exploring code, in regard to professional communication, through the commonly used lens of genre? To investigate our research question, we drew from rhetorical genre studies, particularly as applied to technical and professional communication contexts, to develop a framework for critical analysis that combined study of individual texts (as generic instantiations) and groups of texts (for comparisons across texts). We also used code studies as a basis for theorizing how code may function rhetorically. Based on our rhetorical-textual analysis, we found that code texts whose purposes and structures relate closely enough to suggest recurrence and, thus, generic emergence, possess features of rhetorical activity that warrant further study—study that could benefit practitioners, scholars, and students of technical communication and rhetoric alike. From this article, readers will learn about the activities relating to the development and distribution of genres in code through collaboratively and competitively developed open source software modules. As a result, readers could well expand this initial investigation into broader and more complex bodies of code to understand how professional communication occurs in texts beyond the bounds of conventional discourse. Further, readers may be able to productively exploit this information by working as or alongside programmers to create more purposefully persuasive texts in, as well as in complement to, code.

We begin by investigating, via rhetorical analysis, a collection of related code texts: four spam control modules used in the *Drupal* content management system (CMS). These modules—component parts of the larger CMS—were created

to ensure that only humans can sign up for accounts and use the system. Spam modules are an interesting and important example of how code influences the activities of both developer communities and end users, especially given the ubiquity of site-specific user accounts and the prominence of spam activity that seeks to undermine site stability and user information. As modules with the same essential purpose, these code texts serve as sites where a specific spectrum of typified responses and recurrent action might take place. Other texts, purposes, and genres could be chosen for similar scrutiny, but we believed that spam modules (if not their specific code) and the Drupal framework were both likely easily recognizable to users of Web 2.0 technologies and websites. Further, we felt our own general familiarity with Drupal modules, thanks to our experience as administrators of Drupal sites, would facilitate collection and scrutiny of useful texts as a certain degree of technical competence is required for such an analysis as we propose.

Our primary set of texts for analysis includes multiple versions of each module's code releases, written in the scripting language PHP. We also turn to a secondary set of texts in discursive arguments (in related documentation) about how that code functions rhetorically, found on discussion forums, comment lines in code files, and notes accompanying releases of module versions. These arguments are used by developers to promote particular approaches to project contribution specifically and to the propagation of Drupal spam control modules more broadly, as well as to anticipate potential contexts for use by website administrators and users when the code is incorporated into live Drupal installations. However, while the anticipation of user experiences is significant for our discussion of the rhetorical potential for code and software, when it comes to understanding what rhetorical activity may exist *in code* we attend more fully to how developers work with, and anticipate working with, other developers (explicitly and implicitly) *through the lines of code they write* in order to create potential experiences for users.

Rhetoric, Genre, and Studies of Software

Despite both the last two decade's explosion of computer technologies used in daily life and efforts to draw attention to the underlying infrastructures of these technologies (e.g., Lessig, 1999), the "digital," as a collection of computational machines and the logics and experiences involved in using them, remains obscured for many. More recent efforts to promote digital or computational literacy that have supplemented (or, in some cases, replaced) broader calls for STEM education (e.g., Code.org, 2015). The scope of digital technologies is vast, and the use of the term in various contexts often obscures the carefully constructed logics that underpin digital technologies and systems and their potential for meaningful—and, indeed, *rhetorical*—action. Rhetoric and related fields have enthusiastically considered a vast number of communicative modes

that operate within and surrounding digital activity, from speech and literacy (Cox, 2013; Vee, 2013) to delivery (McCorkle, 2012; Porter, 2009) to procedure (Bogost, 2007; Brock, 2014) to information-seeking and data structures (Beck, 2015; Kelly & Miller, 2016) to software development (Spinuzzi, 2001, 2003b) and broader theoretical concerns about how technologies function rhetorically (Brooke, 2009; Holmes, 2014; Zappen, 2005).

As the act of writing code, or “programming,” is performed by professional and amateur developers—some in this latter group as young as kindergarten students (Code.org, 2014)—for myriad reasons, from critical spacecraft operations to toddler-friendly mobile games. These reasons support both the intended expressive end uses of the programs and the readability of their code for human (developer) audiences. Conventions and values of a development community are continually proposed, tested, challenged, and negotiated by arguments made across instantiations of code—instantiations of typified, protocol-governed practices. However, it has been within the last decade, with the work of Losh (2009), Brown (2015), and Brown and Vee (2016), among others, that critics have investigated how those values are rhetorically embedded and communicated *through* the code composed via development activities, with the majority of studies instead focused on the end-user experience of software or on development activities involving particular software interfaces. Yet meaningful communication among programmers takes the form of computational logic framed through coding style(s), shaped by the needs and constraints of the communities creating and accessing that code and the programs that it comprises.

Scholars studying rhetoric and code have argued that, *as a form of written communication*, code possesses rhetorical features and performs rhetorically in ways both similar to and entirely distinct from features found in more traditional forms of writing (Brooke, 2009; Brown, 2015; Shepherd, 2016). It describes procedures to be interpreted and compiled, transformed and translated into computed (expressed) action by its user(s) as part of some larger activity (Wardrip-Fruin, 2009). As Computer Scientist Donald Knuth, author of the highly influential multi-volume *Art of Computer Programming*, noted in his *Literate Programming* (1992), “The computer programs that are truly beautiful, useful, and profitable must be readable by people. So we ought to address them to people, not to machines” (p. ix). Sharing this goal in their *Elements of Programming Style*, Kernighan and Plauger (1978), as well as Kernighan and Pike in their *The Practice of Programming* (1999), have authored texts on composing code *specifically* in terms of improving its readability and clarity for other human audiences. More recently, following the tradition of Raymond Queneau’s literary experiments, Lopes (2014), a professor of Informatics and Computer Science, has demonstrated the significant effects that various styles of coding may have on the otherwise “identical” resulting program and its output.

Code is both representative of ideas and also executable, but neither form of communication exists as an instrumental medium for the other. As Sample

(2013) observed, “Code may speak to a machine, but it also speaks to us. It is a rich textual object, layered with meaning” (n.p.). We look to code texts in order to learn how rhetorical activity unfolds within coding communities (both formal organizations and looser, ad hoc groups of programmers) as means of creating *social action* by which software developers deliberate with one another about how to code in particular ways toward particular ends. For example, programmers may suggest a preference for certain world-building logics over others *to be applied at multiple scales in their code* (e.g., individual operations, functions, modules) through simple approaches to the organization of conditional case calculations, such as through a reading order of incremental cases (e.g., “A through Z”; whether a kind of *catacosmesis* or *climax*, a sequenced order with specific logics, are followed) or through a structure of most constrained to least constrained cases (e.g., “25,” “any odd number whose value is less than 20,” “all other numbers”; a kind of *expeditio*, a figure of division). World-building logics are not, however, an individual decision and tying debates about these logics, their utility, and how they shape social action must be articulated by looking at a *community* of developers. So that we might consider broader conclusions regarding rhetorical activity in code, we focus our attention here on an overlapping set of communities of developers, their rhetorical situations, exigencies, and recurrent rhetorical responses.

Taking up questions of recurrence (how do similar situations seem to recur?), typification (how do authors respond to these situations in similar ways?), and rhetorical situation (what happens when one or more authors communicate with particular audiences?), we turn to rhetorical genre studies (Miller, 1984). We suggest rhetorical genre studies advances a theoretical conception of genre that can be extended beyond verbal or discourse to help us analyze multiple modalities. Music, television and film studies, fine arts, and a variety of other fields have adopted notions of genre that cross modalities, and the rhetorical conception has been applied to nonverbal modalities, too (for a recent application, see, Ross, 2017). Using the language of genre studies, when we talk about rhetorical genres, we attend less to formal or structural constraint than to the recurrent rhetorical situations and typified responses to those situations (i.e., “types” as kinds of texts that may—but do not necessarily—have formal, as well as rhetorical, features of other texts attempting to achieve similar goals in similar contexts). It is this pairing of form (through typification) and functions (recurrent rhetorical situations) that makes rhetorical conceptions of genre particularly useful to us here. Through this lens, we are able to chart the *social action* a genre performs, offering important insight into rhetorical practices that shape our computational world. We argue that genre is useful for critical rhetorical work in code studies and extend the concept to understand an entirely different modality, code, as rhetorically situated and rhetorically crafted through the processes of design and, of interest to us here, coding (programming).

Rhetorical genre studies have been concerned with the relationship between genres, their text, context, and the role of new media for decades (see, e.g., Yates & Orlikowski's 1992 study of medial change and the emergence of electronic mail). A considerable amount of genre research concerning the role of digital technologies has focused on discourse genres such as the blog (see, e.g., Giltrow & Stein, 2009; Herring, Scheidt, Bonus, & Wright, 2004; Miller & Shepherd, 2004, 2009). Others worked to integrate genre approaches with systems-oriented approaches to study digital ecologies, including activity theory (Spinuzzi, 2003a; 2003b; 2013) and actor-network theory (Kelly & Maddalena, 2015; 2016; Read, 2016). Increasingly, (rhetorical)¹ genre researchers have been attending to the complex new media environments, including the traditional purview in professional and scientific communication, although also exploring new modalities (Casper, 2016; Gonzalez-Pueyo & Redrado, 2003; Gross & Harmon, 2016; Mehlenbacher, 2017; Wickman, 2016). But genre researchers are also investing many other contexts and modalities, including online video genres (including health, e.g., Arduser, 2017 and Ding, 2017; and immigration, Hartelius, 2017), popular technical descriptions (e.g., Pflugfelder, 2017, on Reddit's "Explain to Me Like I'm Five"), video games (e.g., Mehlenbacher & Kampe, 2017; Randall, 2017; Sherlock, 2009), and other popular genres (e.g., Reiff & Bawarshi, 2016).

Scholars have also applied notions of genre to what are typically characterized as technologies, as we wish to do here. Lewis (2016) investigated the user experiences surrounding two technologies, a CMS named Gazelle, and a bittorrent application named Ocelot, to show how "largely invisible server-side software shapes rhetorical action" (p. 4) in order to remind us that "software design is bound to contexts-of-use" (p. 23). When code is written, it is not merely an instrumentally oriented, *ex post facto* transcription of an objective-material reality onto the screen and into the machine's processor (cf. Miller, 1979). Instead, building on arguments made by Brooke (2009), Brown and Vee (2016), and others, we suggest that composing code involves decisions that have rhetorical, epistemic, and ontological consequences. How a program acts, how it is understood to act, and who can or should manipulate it are all concerns that arise out of the program *as it is communicated in its code*. As Russell (1997) has noted, genres "are shared expectations among some group(s) of people. Genres are ways of recognizing and predicting how certain tools [...] in certain typified—typical, recurring—conditions, may be used to help participants act together purposefully" (p. 513). Rhetorical features of code, we posit, exist within its articulation of components (e.g., through naming of variables and functions in code files), through its structure (e.g., by promoting certain procedural logics of how to compute data), and through its process of iterative development (e.g., as multiple contributors create and edit those files).

Given the contemporary social, cultural, and political prominence of software, and particularly of code—to say nothing of the sheer amount of code

written and executed every day, with a 2012 GitHub blog report noting that on a “typical weekday,” over 140 GB of code data is pushed to 125,000 individual software projects hosted on the site (Briandoll, 2012)—we see code as a significant and potent site in which to explore the potential rhetoricity of code. Rhetorical genre studies, especially scholarship on new media genres, is one of many critical lenses through which to discover what conventional rhetorical communication and code can each contribute to an understanding of the other.

Recurrence, Spam, Drupal

A tremendous amount of software is developed under the philosophy of the “open source” movement, referring to a model of software development and distribution where a program’s source code is provided alongside its compiled executable packages for further potential modification by any interested party. Open source is an especially significant approach to software development. The public nature of most open source development allows for easy access to evidence of the competing tensions between individual desires and social constraints; contributions to a given collaboratively developed program (whether in code or documentation) are often debated and revised by numerous members of the development community. If we want to explore what rhetorical activities may be at work in code development and code itself we can most easily do so via publicly accessible open source projects—a kind of convenience sample.

Drupal is an open source software (OSS) application package generally described as a “content management system,” a system that provides infrastructural support for websites from small personal sites to large-scale corporate implementations (Popular Drupal-powered sites include whitehouse.gov, weather.com, nbc.com, and bbc.com). We chose Drupal because it is a widely recognizable and complex open source software application with a large (107,442 users; reported by Drupal in June 2017) and diverse (groups include the African federation, the Kochi group, Drupal North Regional Summit, the Hunsrück, and more; Drupal, 2017) developer base. Certainly, a number of open source platforms would likewise be suitable for further study, but the popularity of this platform and the authors’ familiarity with its technical functioning allowed for both a widely recognizable object of study and one the authors have the expertise to discuss.

Drupal *modules* allow Drupal installations to be modified to suit specific contexts and needs, such as the prevention of spam account and content creation that might otherwise overwhelm a given site. Some modules are written by the core Drupal development community and others written by third-party communities, which provides an interesting range in the community of developers.

We are interested in modules that see active development as we hope to chart how conventions operate across development cycles. To that end, we are interested in spam modules, given the widespread need for continued improvement

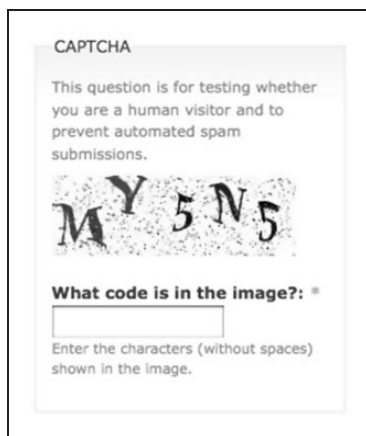


Figure 1. CAPTCHA test and submission form.

(and thus development) in security protocols to protect websites of all kinds, especially those—like sites built on Drupal—that make use of a database for data storage and recall. Spam and spam control technologies evolve together in a continuous war of escalation over which can counteract the other more effectively (for a detailed history of spam and its antidotes, see Brunton, 2013); as various some anti-spam approaches become ineffective and thus obsolete, there arise opportunities to develop new spam control technologies in response to or anticipation of emerging spam types. One popular approach to spam control is CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). Specific CAPTCHA methods vary in implementation—see Figures 1 and 2 for two variations of the test—but the underlying principle is the same to determine from a set of user input data if that user is human or software (commonly referred to as a “spam bot”).

Multiple CAPTCHA Drupal modules have been developed over the course of the last decade and have innovated spam detection and prevention, such as by increasing the difficulty for nonhumans to respond to tests—such as by obscuring test text in images that users must input as responses (to fool optical character recognition, or OCR, software, which discerns text characters within image files) or by asking questions that are easy for a human possessing contextual knowledge to answer (e.g., “What is the three-letter abbreviation for the United States of America?”).

These related modules, we suggest, serve as useful examples of texts that could be understood as instantiations of a rhetorical genre in code—that is, they are marked by a recurrent rhetorical situation and by typified structures and algorithmic procedures (cf. Miller, 1984). To illustrate how these modules, these computational objects, are rhetorical and typified, we investigate four related Drupal spam prevention modules: CAPTCHA, reCAPTCHA, Captcha



Figure 2. ReCAPTCHA test and submission form.

Riddler, and BOTCHA. These four modules were chosen in part for the popularity of CAPTCHA and in part for their distinct mutations of CAPTCHA’s basic functionality. These modules operate rhetorically through their code texts for developer audiences, through the logics of their tests for discerning human from computer and the stylistic implementation thereof in code files. In addition, the modules work rhetorically through their anticipated action (i.e., their use) by website administrators and users who may never look at the code but whose use of Drupal is nonetheless impacted by it, thanks to the explanations provided by each module’s authors about how it should function and how it is integrated into a site that uses it.

The *CAPTCHA*² module is described as a “challenge–response test most often placed within web forms to determine whether the user is human,” with the primary goal being to block automated submissions designed to spam

websites with ads (Wundo, 2004). The project has been maintained by 49 people over the last 10 years, with 641 commits to the code base, 1,334,072 downloads, and 245,342 installs.³ The following is a simplified explanation of the CAPTCHA workflow:

1. When a page with a CAPTCHA prompt is requested by some user (whether human or nonhuman), such as a login page, generate a unique string of alphanumeric characters and store that string.
2. Using that stored string, generate a multilayered image to display the string in a manner that will hopefully be difficult for OCR technologies to read (e.g., using character-warping distortion, additional lines over the text, and visual salt-and-pepper noise to obscure character shapes).
3. Provide the user with an input field to enter the text string that he/she/it believes matches the stored string displayed in the generated image.
4. Check the input string against the stored string.
5. If the check is successful, then allow the user's action to be successful. If the check is unsuccessful, then prevent the user from completing his/her/its intended action.

ReCAPTCHA, one of the most prominent descendants of CAPTCHA, is described as intending to “improve the CAPTCHA system and protect email addresses” (RobLoach, 2007). A project developed by employees at Google, the most widely used versions of ReCAPTCHA employ data from the Google Maps databases to generate their validation tests, while the newest iterations of the module rely on contextual knowledge (e.g., expecting users to click on appropriate images from a larger set of images, such as “Select all soup”). Previously, it used validation input to help clarify text in Google Books’ digital archives that had been difficult for OCR technology to discern. ReCAPTCHA has been maintained by 13 people over the last 8 years, with 165 commits, 548,413 downloads, and 64,664 installs. (It is worth noting that CAPTCHA and ReCAPTCHA are the two most downloaded Drupal modules for any security-related purpose; Table 1 shows comparable statistics about these discussed modules, along with the third, fourth, and fifth most-downloaded security modules: OAuth, Access Control Lists, and Search Configuration.)

Captcha Riddler allows “site administrators create their own questions to foil automated spam bots” rather than using randomly generated strings of characters for verification (Imerlin, 2007). These questions can be customized to be more or less exclusive in determining “real” users through either contextually appropriate or entirely arbitrary knowledge, depending on the desires of a given site’s administrators. Since May 2007, 5 developers have contributed 32 commits to the project, garnering 13,298 downloads and 4,527 installs.

Finally, *BOTCHA* is a “non-CAPTCHA spam protection framework” whose functionality is sometimes categorized as a “honeypot.” BOTCHA attempts

Table 1. Spam Prevention Modules and Other Most-Downloaded Security Modules.

Project	Year begun	Commits	Maintainers	Downloads	Installs	Actively maintained?
CAPTCHA	2004	641	49	1,334,072	245,342	Yes
ReCAPTCHA	2007	165	13	548,413	64,664	Yes
Captcha Riddler	2007	32	5	13,298	4,257	Maintenance fixes only
BOTCHA	2010	249	6	24,074	8,202	Maintenance fixes only
OAuth	2008	364	23	410,894	65,942	Yes
Access control lists	2006	147	6	217,761	24,373	Yes
Search configuration	2006	64	9	72,377	19,073	Yes

to identify spam bots by hiding tests from users that it thinks are humans; as tests are completed by a user, the system can more accurately determine whether that user should be exposed to further tests and restricted from actually using the site (Iva2k, 2010). Four years old, BOTCHA has 6 maintainers and 249 commits and has generated 24,074 downloads and 8,202 reported installs; despite its relatively recent creation, there has been some considerable uptake of the module, especially in comparison with Captcha Riddler.

The numbers involved for each module help suggest the generic nature of spam prevention modules; there is considerable activity taking place to typify this sort of software (i.e., as “spam prevention”), with differing levels of contributor commitment and user (website administrator) uptake—with enough downloads and installs occurring to suggest recurrence of similar rhetorical situations and user needs.

Modules as Genres Typified in Information Design

Each module includes code blocks that perform relatively uniformly for the core system, loading conventional information such as module-related help, theme (appearance) details, and so on. Significant changes to *these* aspects of each module’s code occur primarily in response to changes to Drupal’s core architecture rather than to any changing needs or preferences of the module’s developer community. In the main `.module` file for each project, Drupal functions are included first, and only after that does any code unique to each project appear, further emphasizing the relationship between large-scale constraint and small-scale individual-project preference.

Although these modules together contribute toward the construction of emerging genres related to spam control, they also perpetuate a tradition of several

antecedent genres (Jamieson, 1975), most notably (a) at the level of language, as code texts written in PHP and thus leaning heavily on 20 years of emergent idiomatic language use and (b) at the level of application, namely as third-party plugins for Drupal. At both these levels, these modules flow generically through related conceptual “streams” (i.e., as part of the distribution of Drupal or the spectrum of uses for the PHP language) of distribution, through which developers can interact procedurally with diverse user and administrator populations (Lena & Peterson, 2008).

The collection of code texts written in any given programming language serve as a broad typified collection; as the language affords particular data structures, functional or object-oriented procedural logics, and so on, clusters of recurrent practices emerge as temporarily dominant paradigms, with their use a combination of current “best practices” and varied iterations of available examples. Spinuzzi (2002), in a discussion of genre knowledge among programmers, asserts that any given block of code “gives programmers an idea of what the original programmer’s intention was and helps them to refine their guesses about how this chunk of code helps to meet the program’s goals” (n.p.). Their guesses may not be identical to that original intention or motive, but a reliance on generic conventions, such as contemporary values about best practices, almost certainly assists with their interpretative process. Further, this act of reflective interpretation may provide developers with the opportunity to align their own individual motives with those anticipated to be held by others, as well as to reify or challenge the values perceived to be held by members of the relevant development community or organization.

Drupal modules similarly provide constrained freedom for developers to pursue certain ends with their projects so long as those projects, in code and expression, support the projected trajectory of development for the core Drupal system. That is, a module whose developer is less likely to pursue highly customized Drupal architecture to facilitate its use will probably experience more uptake than a module whose developer attempts to create a more customized architectural scheme.

As code files written in PHP, these modules conform to the constraints of the formal language that translates their authors’ procedural logics into machine-readable commands. In one sense, there is little room for creative exploration of PHP’s affordances—code has to be “comprehensible” to a web server in order for it to be expressed as intended. However, the range of avenues by which a developer might construct a particular program offers that developer, and his or her fellow contributors, opportunities to approach their code not, as Burgess (2010) has argued, as “the insertion of content, but the beginnings of an understanding of logical structuring” about how a program works (p. 171). Emergent genres of code practice in PHP then are constrained not only in regard to what the code might potentially allow for *mechanically* but also in regard to what a human reader who has access to the code (since PHP is rendered by the web

server, not the client browser) *might understand about how it could or should function*, what sorts of procedural components or operations should be valued or disregarded in its application, and so on.

These modules also possess similar structural qualities imposed upon them by the overlapping communities that use and develop module projects for the CMS, such as specific file, function, and variable organization and naming conventions. Particular constraints announce a set of rhetorical efforts and goals to Drupal developers and system administrators (i.e., that module code can be uploaded into specific subdirectories and functionally manipulated through Drupal's browser-based administrative interface). Those audiences can easily identify what each project "is" and what it "does" (providing some specific functionality for a given system) through these qualities. For example, `[module name].module` and `[module name].info` are files required for every module, and they provide certain information about how that module works: The former provides the code for its expression through Drupal; the latter provides information about the module's version, its general functionality (e.g., spam control), and what other package dependencies it relies upon to function.

Although three of the four modules are structured in a similar manner (likely due to their common basis in CAPTCHA), BOTCHA stands apart as having a different, "exploded," organizational structure. It follows a paradigm common to many contemporary non-Drupal software interfaces, that of "model/view/controller" or (MVC)—comprised of a model that manages data in an application, a view that represents output data provided by the model, and a controller that manipulates the model based on input data—that distinguishes the roles of its major code components for easier developer navigation and conceptual abstraction of each (Stolley, 2015). In contrast, the other CAPTCHA modules far more closely resemble one another—and *the structure of the core Drupal modules*—likely due to their shared origin in the main CAPTCHA project, which as the oldest of these modules has its own origin squarely within the Drupal organizational paradigm.

Modules as Genres Typified in Code

There are two major audiences for Drupal modules' code genres, and they differ in regard to the "delayed" or potential action of how the code texts persuade through their composition and (machinic) interpretation. The more obvious of the audiences for the modules, broadly speaking, are end users and the administrators of individual Drupal installations: Individuals in either of these roles engage a module in its enacted form (e.g., having to pass the CAPTCHA test to determine whether he or she should be authorized to post content on a Drupal site). However, in regard to the modules as genres in *code*, it is the members of Drupal development communities that serve as the primary users of these

genres—both in anticipation of future user-oriented activities (such as those described above) *and* in regard to immediate development-related communication.

Developers deliberate with their colleagues in and around code (such as in comment lines or in e-mails or discussion forums) about how best to anticipate those activities they expect to happen when users encounter the modules, whether directly or indirectly. Particular constructions of code blocks facilitate, more or less usefully than alternate constructions, particular approaches to realizing a given activity. Further, these constructions suggest preferences among individual or group developers about how the module *should* work to perform its intended function(s).

This sort of deliberation makes use of rhetorical strategies that, in code, do not always closely resemble the strategies that we identify easily in more conventional forms of discourse. There are certainly some moves that may seem familiar; for example, style consistency (throughout one or more files in a given module, or across multiple similar kinds of modules) serves as a means of demonstrating to one's colleagues that one may be attempting to develop a recognizable grammar for readability purposes as well as solving (what are perceived to be) similar kinds of problems through recurrent operations or functions. Similarly, logical structures of certain operations—such as locating the appropriate values of data in specific array positions—work to suggest a particular means of generic identification and formation.

However, the ways that logical procedures perform arguments in code can be radically different from how we expect logical arguments to be constructed discursively. One does not necessarily establish a claim or its warrants identically in imperative code languages the way one might in natural spoken or written languages. Our distinguishing between arguments made in *conventional discourse* and those made *in code* might seem initially confusing, not because one might expect them to work identically but because one might be initially unwilling to accept the existence of the latter. Generally, we expect procedure (of all kinds, not just code specifically) to function as the skeleton for an argument to be expanded and clarified considerably by explanation and example, but we do not necessarily expect procedure itself to *be* an argument. However, as Chun (2008) has noted, code texts and executable software are not identical, but rather perform rhetorically in unique, although connected, ways that create different kinds of meaning; for Chun, source code as a site for investigation “is arguably symptomatic of human language’s tendency to attribute a sovereign source to an action, a subject to a verb” (p. 309). Although we do not want to ascribe the site of *all* meaningful activity to the code text from which a software program is constructed, neither do we want to ignore its contributions to how developers and other readers construct and interpret meaning from the code text as it is examined, debated, and revised. Accordingly, the kinds of meaning-making that one encounters in code, when acknowledged, often *appear* to be highly implicit

but nonetheless play significant roles in communicating meaning throughout a development community.

For example, in Table 2, the same essential functionality—how a CAPTCHA-related module generates its basic spam test—is constructed in two distinct but closely related ways and thus each communicates important information about its purpose and execution. The default CAPTCHA test is a simple mathematics question that asks a user to provide the solution for the equation “ $x + y$ ” where “ x ” and “ y ” are two numbers randomly generated and displayed on the screen, with the value of each ranging from between 1 and 20. The expected answer (saved as the variable `$answer`) is then compared against the user’s input (saved as `$result['solution']`). The code for Captcha Riddler is structured syntactically in a very similar manner and performs the same essential function, displaying a prompt and expecting a solution from a database of existing question–answer pairs. In part, this may be due to the developers of Captcha Riddler wanting to imitate the structure of CAPTCHA, for example, employing the same essential variables and test generation structures (which could, then, facilitate future contributions from developers of other CAPTCHA-related modules). Regardless of intent, what we can see is a set of texts that, in their similar approach to establishing how a spam test could and should be generated, how input is constrained (as a one- or two-digit

Table 2. Excerpts From Captcha_Captcha() and Riddler_Captcha() Module Hooks.

Line	From CAPTCHA, captcha.module (version 7.x-1.12)	Line	From Captcha Riddler, riddler.module (version 7.x-1.2)
731	function captcha_captcha(\$op, \$captcha_type = '') {	224	function riddler_captcha(\$op, \$captcha_type='') {
...		...	
736	case 'generate':	229	case 'generate' :
737	if (\$captcha_type == 'Math') {	230	if (\$captcha_type == 'Riddler') {
738	\$result = array();	231	\$result = array();
739	\$answer = mt_rand(1, 20);	232	\$riddles = riddler_get_riddles();
740	\$x = mt_rand(1, \$answer);	233	\$key = array_rand(\$riddles);
741	\$y = \$answer - \$x;		
742	\$result['solution'] = "\$answer";	234	\$result['form']['captcha_response'] = array(235 '#type' => 'textfield', 236 '#title' => \$riddles[\$key] ['question'], 237 '#description' => t('Fill in the blank.'), 238 '#size' => 50, 239 '#maxlength' => 50, 240 '#required' => TRUE, 241 '#weight' => variable_get('riddler_weight', 0), 242); 243 \$result['solution'] = (string) (drupal_strtolower(\$riddles[\$key] ['answer'])); 244 \$result['captcha_validate'] = 'riddler_captcha_validate'; 245 return \$result; 246 }
...			
747	\$result['form']['captcha_response'] = array(248 '#type' => 'textfield', 249 '#title' => t('Math question'), 250 '#description' => t('Solve this simple math problem and enter the result. E.g. for 1+3, enter 4.'), 751 '#field_prefix' => t('@x + @y = ', array('@x' => \$x, '@y' => \$y)), 752 '#size' => 4, 753 '#maxlength' => 2, 754 '#required' => TRUE, 755); 756 return \$result; 757 }		

number for CAPTCHA, or as alphanumeric characters transformed to lower-case for Captcha Riddler), and how the user's input is then compared with the Drupal-provided answer.

As with its demonstration of genre membership or association in regard to information design, each module's code, in its composition, articulation, and expression, could suggest possible tensions between the service of Drupal-level rhetorical goals and those goals unique to the module's development community—just as individual installations of each module might be customized to specific sites but would need community support for those customizations to be accepted in future versions of the published module. “Tension” between what Bakhtin identified as the “centrifugal and centripetal” (decentralizing and centralizing) forces at work on a genre, which Schryer (1999) built upon to offer use a definition of genres as “stabilized-for-now or stabilized enough sites of social and ideological action,” offers a reminder that genres are not fixed (p. 81; see also, Schryer, 1993, 1994). Rather, the “tension” is between the users and the abstraction we call “genre,” the former exerting individual influence (assisting in a decentralizing) while the latter exerts collective typification (a kind of centralizing). Returning to Miller's (1984) list of features of rhetorical genres, any potential tensions could find purchase in the module-as-genre because a genre “mediat[es] private intentions and social exigence; it motivates by connecting the private with the public, the singular with the recurrent” (p. 163). The module becomes a site for experimentation and customization as well as for organizational uniformity and stylistic consistency as various authors and relevant communities determine, with varying degrees of success, how best to make their cases through new iterations of and modifications to the genre. This can be witnessed to an extent in comparisons of file versions showing changes made to modules' code over time; some changes appear to occur in response to new contributors' stylistic preferences as well as to mechanical demands and constraints imposed by significant shifts in Drupal infrastructure from one major version to the next. If the developers of each module cannot completely avoid Drupal-imposed constraints that might be otherwise undesirable to them for the module's goals, they can at least articulate their preferences for alternatives provided in both its code and its comments.

Broadly, we perceive each of these modules demonstrating increasingly localized modular distinctions of code operations, such as the separation of test generation from validation, or the transformation of individual variables into more complex multielement arrays to facilitate looping, iterative processes of data manipulation. The increase in modularity could suggest a number of rhetorical goals. First, this might indicate an anticipation of the growing complexity of a given project, with the modular “explosion” of its code avoiding potentially unwieldy or complex functions. Second, it might reflect anticipated vectors of future expansion for the project—regardless of whether these are ultimately pursued and realized by the community. Third, particular developers might

find themselves adopting or assuming control or authority over modular components of which they are particularly fond, potentially leading those components to become entirely independent projects from the originating module.

More specifically, ReCAPTCHA and Captcha Riddler plug into the intermediary framework of CAPTCHA to function successfully as spam prevention. Specifically, they each connect to CAPTCHA's utility through a subordinate relationship that the Drupal community defines as a "hook":

Drupal's module system is based on the concept of "hooks." A hook is a PHP function that is named `foo_bar()`, where "foo" is the name of the module (whose filename is thus `foo.module`) and "bar" is the name of the hook. Each hook has a defined set of parameters and a specified result type. (Hooks, n.d.)

As a result, each is constrained in its users' expectations for it to perform within this larger framework; each module innovates only to the extent that it does not "break" the use of CAPTCHA for spam control, for example, the question–answer validation schema. The naming structure for each module's central testing function (named `[module name]_captcha()`, e.g., `recaptcha_captcha()`) supports this subordinate relationship of each module to CAPTCHA, that is, that each module is itself an additive component that hooks into the expressive functionality of the central CAPTCHA utility.

BOTCHA is exceptional in this regard, as it operates separately from the operative model of CAPTCHA and its connected modules. Because of this distinction, BOTCHA's developers frame their validation tests differently, as *recipes* to be customized as individually desired, constructed for particular validation tests in specific conditions (e.g., when a user attempts to input its password as part of the login process). Where a particular CAPTCHA test can be assigned to a particular condition, BOTCHA offers the ability to mix validation efforts with somewhat more freedom, in order to reduce more effectively the potential for spam bots to gain access to a system. However, because BOTCHA operates most fundamentally in a manner similar to the CAPTCHA-related tests through its comparison test (of generated and input text), its code and activity demonstrate the breadth of the genre's, we might say, potential for responding to spam.

Understanding These Modules as Rhetorical Genres

So what does it mean that each of these modules—and the group of modules collectively—represents a *rhetorical*, and not simply a taxonomic, genre? Further, what might the possibility of identifying and exploring critically genres or genre-ing activity in code suggest more broadly for potential avenues of inquiry in rhetoric?

First, that each of these modules is developed in response to the same recurring situation or problem suggests that this situation is potentially understood

by developers to be an exigence in need of a resolution. In this case, that exigence is the abundance of spam bots attempting to gain access to a Drupal site's content production tools (usually to publish links to sites that install malware, generate revenue for their administrators via click-through activities, or both); the resolution, in turn, is the prevention of such access. Further, an issue of content control impacts the continued maintenance of a given site (and thus any subsequent user activity on it, as well): Automated spam prevention facilitates an administrator's work by not having to continually purge his or her site's database of spam content or accounts. This is rhetorically significant because the presence of spam marks an effort to persuade one to leave one site for another, likely in order to sell one some product or compromise the security of one's computer or user account. Perhaps more importantly, spam can significantly undermine the credibility of a site, orienting an audience away from the content original to the website in frustration or even fear of security breaches. Developers' awareness of this exigence—and responses to it as working *generically*—could be investigated in more, and more empirical, depth through other methods of analysis (e.g., surveys or interviews).

Second, the development of a potential resolution, a typified response, through the composition of modules whose functional components resemble those of similar modules suggests that developers understand how to respond to that exigence most optimally in a particular, and shared, set of ways. It is through this approach that a typified response emerges: A string or other set of data is generated and stored by the site to be compared against the input of a suspected spam bot account. There are certainly other ways an administrator might work to stop spam (whether as a variation on the basic CAPTCHA theme, such as that demonstrated by the operation of BOTCHA, or through some entirely distinct, unrelated method of spam prevention that other modules use).

In other words, the developers of each module approach their perceived exigence with a rhetorically informed grasp of procedural logic. Their activities resemble one another closely enough to allow us to see the formation of a genre: In its code files, each spam prevention module is demonstrated through the composition of a recurrent set of functions, each of which executes a particular operation to facilitate the module's goal. This recurrence may be perceived most clearly through the reliance on the `[module name]_captcha()` hook described earlier, since each hook performs a similar set of operations—albeit with different specific parameters, test generation “types,” and so on. See the earlier discussion of Table 2 for a comparison of rhetorically and functionally similar lines of code from CAPTCHA and Captcha Riddler.

Across these different modules, the goals and mechanisms of these module hooks perform closely comparable rhetorical acts. The goals of these code lines are realized through relatively similar means, namely the setting of boolean and string data variables, as well as the comparison of boolean and string data

variables that will determine the conditional parameters to be activated elsewhere in each module's code. In each case, a developer can reasonably assume that `[module]_captcha()` contains lines of code that will

- have recognizable names when appropriate (for other developers to recognize the purpose of specific functions and variables),
- access a database of generated test values to populate the spam test,
- display a generated value for the user to replicate (and thus confirm his or her nonspam nature), and
- compare the input value with the generated value.

Further, in each case, according to the logic of the module's preferred method of generating tests, a developer can expect to see lines of code that attempt to realize that method in a relatively concise and readable number of operational steps; this pair of qualities is commonly referred to as "elegance" for such code's ability to rhetorically and functionally meet its goals in an aesthetically pleasing way. Similarly, generic recurrence of these qualities facilitates continued experimentation with effective approaches toward rhetorical communication of these code operations, which might result in potential cross-module development or further iterative mutations of this sort of generated test (whether for spam prevention, as in the case of BOTCHA reconceiving how the CAPTCHA test is deployed for certain users, or for even more diverse ends). Spinuzzi (2002) suggests that, given the nature of code as mechanical processes, a programmer may eventually achieve "complete" knowledge of a program in a way that is impossible for interpretation to facilitate. We suggest that, if code is explored rhetorically as a type of meaningful communication, then—like other forms of communication—such knowledge is rarely, if ever, as complete as an instrumental understanding thereof might suggest. Nonetheless, expanding our knowledge of how code *acts* rhetorically as well as mechanically might well lead to deeper and more nuanced engagements with, and uses of, code for any number of ends.

Final Remarks

As genred activity in code, the Drupal CAPTCHA modules considered in this essay demonstrate both considerable variety and recurrence in the procedural logics comprising their test generation and validation. Although these tests have not changed much in any case, each project has, over time, increased the scope and variety of the implemented security measures surrounding its key generation procedures (e.g., restricting access to variables and data in the SQL database; setting and flushing individual session ID information; cascading multiple validation tests) as a way to establish the project as a reliable and trustworthy module for spam control.

In addition, these modules as code texts form a complex genre system, which Bazerman (1994) described as a set of genres that facilitate “the full interaction, the full event, the set of social relations as it has been enacted” via “intertextual occurrences [. . . that attend] to the way that all the intertext is instantiated in generic form establishing the current act in relation to prior acts” (p. 99). For the module genre system, so-to-speak, typifications in code like this CAPTCHA-oriented spam prevention code interact with—and are fundamentally influenced by—such interrelated genres of software development (which also possess counterparts as activities) as: Drupal core module, spam prevention software, software written in the PHP language, web server and browser software, and the discourses used while collaboratively developing code. Just as none of these modules exist in a contextual vacuum, neither does the code for it operate without being informed and shaped by software that intersect with it, software which themselves run on other assemblages of code for particular and distinct ranges of anticipated use and development practices. As Miller and Shepherd (2009) noted in regard to the transformation of the blog from genre to medium, “what persists is not only the medium but also the recurring rhetorical forms that it enables, or even requires (such as, for blogging, reverse chronology, links, comments, present tense, brevity), as well as the audience expectancies that these forms promote (such as frequent updating, connection, authenticity)” (p. 284). Similarly, code texts serve simultaneously as media—for the construction of interpretable or executable software—as well as demonstrations of communicative genres that work persuasively for their authors and development audiences alike, drawing together like-minded programmers while also reflecting individual motivations to push continued work on certain projects in particular directions.

If we claim that, as technical communication researchers, we are interested in the rhetorical potential of emerging technologies, then we must attend further to how the mechanisms of those technologies are communicated, perpetuated, and changed over time *through their written and expressive media* (code for, as well as use of, software) by the communities of developers that engage them directly rather than focusing solely on end-user experiences potentially unaware of but impacted by developers’ procedural arguments. That is, it is important that we continue investigating how code and algorithmic procedure shapes users’ experiences, and to better do that as technical communicators, we must continue theorizing the role of code as not only as technology but as rhetorical work as well.

What open source software (OSS) programming offers us, then, is a continued opportunity to study the complex rhetorical environments in which development occurs; in this, we echo—among others—Johnson-Eilola (2002) and others since (such as Cripps, 2011; Maher, 2011; Qiu, 2016) who have argued for technical communicators and critics to contribute to the OSS movement by developing documentation informed by (among others) rhetoric, usability theory, and

information design. Situating documentation such as README files, man pages, help files, and so forth, as well as the code itself for any given OSS project, are open—that is, publicly available—for researchers to access and analyze. Further, in many cases, project repositories for documentation and software hosted on sites like GitHub provide file revision histories, so we might track both development and commentary on the development process. Such sites are often rich with the rhetorical strategies developers employ to advance their cause, and these engagements can tell us much about how collaboratively developed software is shaped, as well as the kinds of arguments, expertise, and values that contribute to such crafting. Similarly, we recognize the impact that default settings and the ideological underpinnings of those settings have on user behavior, community makeup, and social policy (Shah & Kesan, 2008). Technical communication experts, further, might benefit the communities they study with their own expertise, pedagogically informed approaches to collaborative writing and design, and insight into crafting better, and more open, documentation for and in collaboration with the growing number of developers participating in these projects.

Students of technical communication, rhetoric, and composition may thus find useful the study of code as not merely a vehicle for, but a site and means of, rhetorical activity. By understanding code as such, enterprising students might well more effectively influence—whether directly as programmers or indirectly as technical communicators working alongside programmers in the design process—the development of software with explicitly persuasive code that communicates its functionality, and the intent of its authors, more clearly to current or future collaborators. Such an approach does not necessarily require programming expertise on the part of aspiring technical communicators, but it would demand they understand how to employ procedural logic for complex computational ends. Further, talking about code in terms of genre-ing activity helps teach students to approach the construction of software, and the code texts that comprise them, as a process where the rhetorical situation shapes the kind of response(s) that can be drawn from a highly typified set of responses, which is a useful analogy for us to deploy as we continue the tradition of interrogating rhetoricity of artificial languages.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Notes

1. Hyon (1990) provides an overview of three well-established traditions in “genre studies”: Rhetorical Genre Studies, Systemic Functional Linguistics, and English for Specific Purposes. Emerging schools of genre are also developing, including in Brazil and Denmark (Artemeva, 2016; Miller & Kelly, 2016).
2. Web addresses for the four modules we examine are CAPTCHA at <https://www.drupal.org/project/captcha>, ReCAPTCHA at <https://www.drupal.org/project/recaptcha>, Captcha Riddler at <https://www.drupal.org/project/riddler>, and BOTCHA at <https://www.drupal.org/project/botcha>.
3. This information was collected by the authors on January 14, 2015. Number of contributors, commits, and project descriptions may change, but there remain interesting trends, noted in the body of this article, included by this snapshot of maintainers and project contributors.

References

- Albers, M. J. (2009). Design for effective support of user intentions in information-rich interactions. *Journal of Technical Writing and Communication*, 39(2), 177–194.
- Arduser, L. (2017). Remediating diagnosis: A familiar narrative form or emerging digital genre? In C. R. Miller & A. R. Kelly (Eds.), *Emerging genres in new media environments* (pp. 63–78). London, England: Palgrave Macmillan.
- Artemeva, N. (2016). *Genre studies around the globe: Beyond the three traditions*. Victoria, BC: Trafford Publishing.
- Bazerman, C. (1994). Systems of genres and the enactment of social intentions. In A. Freedman & P. Medway (Eds.), *Genre and the new rhetoric* (pp. 79–101). London, England: Taylor and Francis.
- Beck, E. (2015). The invisible digital identity: Assemblages in digital networks. *Computers and Composition*, 35(1), 125–140.
- Birkbak, A., & Carlsen, H. B. (2016). The world of Edgerank: Rhetorical justifications of Facebook’s news feed algorithm. *Computational Culture*, 5. Retrieved from <http://computationalculture.net/article/the-world-of-edgerank-rhetorical-justifications-of-facebooks-news-feed-algorithm>
- Bogost, I. (2007). *Persuasive games: The expressive power of videogames*. Cambridge, MA: MIT Press.
- Briandoll. (2012). The Octoverse in 2012. *GitHub*. Retrieved from <https://github.com/blog/1359-the-octoverse-in-2012>
- 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/>
- Brooke, C. G. (2009). *Lingua fracta: Towards a rhetoric of new media*. New York, NY: Hampton Press.
- Brown, J. J. Jr. (2015). *Ethical programs: Hospitality and the rhetorics of software*. Ann Arbor, MI: University of Michigan Press.
- Brown, J. J., Jr., & Vee, A. (2016). Rhetoric special issue editorial introduction. *Computational Culture*, 5. Retrieved from <http://computationalculture.net/editorial/rhetoric-special-issue-editorial-introduction>
- Brunton, F. (2013). *SPAM: A shadow history of the Internet*. Cambridge, MA: MIT Press.

- Burgess, H. J. (2010). <?php>: “Invisible” code and the mystique of Web writing. In B. Dilger & J. Rice (Eds.), *From a to <a>: Keywords of markup* (pp. 167–185). Minneapolis, MN: University of Minnesota Press.
- Casper, C. F. (2016). The online research article and the ecological basis of new digital genres. In A. G. Gross & J. Buehl (Eds.), *Science and the internet: Communicating knowledge in a digital age* (pp. 77–98). Amityville, NY: Baywood.
- Chun, W. H. K. (2008). On “sourcery” or code as fetish. *Configurations*, 16(3), 299–324.
- Code.org. (2014). Teach our K-8 intro to Computer Science. *Code.org*. Retrieved from <https://code.org/educate/20hr>
- Code.org. (2015). Anybody can learn. *Code.org*. Retrieved from <https://code.org/>
- Cox, G. (2013). *Speaking code: Coding as aesthetic and political expression*. Cambridge, MA: MIT Press.
- Cripps, M. J. (2011). Technical communications in OSS content management systems: An academic institutional case study. *Journal of Technical Writing and Communication*, 41(4), 423–448.
- Ding, H. (2017). Cross-culturally narrating risks, imagination, and realities of HIV/AIDS. In C. R. Miller & A. R. Kelly (Eds.), *Emerging genres in new media environments* (pp. 153–170). London, England: Palgrave Macmillan.
- Giltrow, J., & Stein, D. (2009). *Genres in the Internet: Issues in the theory of genre*. Amsterdam, The Netherlands: John Benjamins.
- Gonzalez-Pueyo, I., & Redrado, A. (2003). Scientific articles in Internet homepages: Assumptions upon lay audiences. *Journal of Technical Writing and Communication*, 33(2), 165–184.
- Gross, A. G., & Harmon, J. E. (2016). *The Internet revolution in the sciences and humanities*. Oxford, England: Oxford University Press.
- Hartelius, E. J. (2017). Sentimentalism in online deliberation: Assessing the generic liability of immigration discourses. In C. R. Miller & A. R. Kelly (Eds.), *Emerging genres in new media environments* (pp. 225–241). London, England: Palgrave Macmillan.
- Herring, S. C., Scheidt, L. A., Bonus, S., & Wright, E. (2004). Bridging the gap: A genre analysis of weblogs. In R. H. Sprague Jr. (Ed.), *Proceedings of the 37th Annual Hawaii International Conference on System Science* (pp. 101–111). Los Alamitos, CA: IEEE Computer Society Press.
- Holmes, S. (2014). Rhetorical algorithms in Bitcoin. *Enculturation: A Journal of Rhetoric, Writing, and Culture*, 18. Retrieved from <http://enculturation.net/rhetoricalalgorithms>
- Holmes, S. (2016). ‘Can we name the tools?’ Ontologies of code, speculative techné, and rhetorical concealment. *Computational Culture*, 5. Retrieved from <http://computationalculture.net/article/can-we-name-the-tools-ontologies-of-code-speculative-techné-and-rhetorical-concealment>
- Hooks. (n.d.). *Drupal API*. Retrieved from <https://api.drupal.org/api/drupal/includes%21module.inc/group/hooks/7>
- Hyon, S. (1996). Genre in three traditions: Implications for ESL. *TESOL Quarterly*, 30(4), 693–722.
- Imerlin. (2007). Captcha Riddler. *Drupal*. Retrieved from <https://www.drupal.org/project/riddler>
- Iva2k. (2010). BOTCHA Spam Prevention. *Drupal*. Retrieved from <https://www.drupal.org/project/botcha>

- Jamieson, K. M. (1975). Antecedent genre as rhetorical constraint. *Quarterly Journal of Speech*, 61(4), 406–415.
- Johnson-Eilola, J. (2002). Open source basics: Definitions, models, and questions. *SIGDOC '02*, 79–83.
- Kelly, A. R., & Maddalena, K. (2015). Harnessing agency for efficacy: 'Foldit' and citizen science. *POROI- The Project on Rhetoric of Inquiry*, 11(1), 1–20.
- Kelly, A. R., & Maddalena, K. (2016). Networks, genres, and complex wholes: Citizen science and how we act together through typified text. *Canadian Journal of Communication*, 41, 287–303.
- Kelly, A. R., & Miller, C. R. (2016). Intersections: Scientific and parascientific communication on the Internet. In A. G. Gross & J. Buehl (Eds.), *Science and the internet: Communicating knowledge in a digital age* (pp. 221–245). Amityville, NY: Baywood Press.
- Kernighan, B. W., & Pike, R. (1999). *The practice of programming*. Reading, MA: Addison-Wesley.
- Kernighan, B. W., & Plauger, P. J. (1978). *The elements of programming style* (2nd ed.). New York, NY: McGraw-Hill.
- Knuth, D. E. (1992). *Literate programming*. Stanford, CA: Center for the Study of Language and Information.
- Lena, J. C., & Peterson, R. A. (2008). Classification as culture: Types and trajectories of music genres. *American Sociological Review*, 73(5), 697–718.
- Lessig, L. (1999). *Code and other laws of cyberspace*. New York, NY: Basic Books.
- Lewis, J. (2016). Content management systems, Bittorrent trackers, and large-scale rhetorical genres: Analyzing collective activity in participatory digital spaces. *Journal of Technical Writing and Communication*, 46(1), 4–26.
- Lopes, C. V. (2014). *Exercises in programming style*. Boca Raton, FL: Chapman and Hall/CRC Press.
- 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.
- 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.
- McCorkle, B. (2012). *Rhetorical delivery as technological discourse: A cross-historical study*. Carbondale, IL: Southern Illinois University Press.
- Mehlenbacher, A. R. (2017). Crowdfunding science: Exigencies and strategies in an emerging genre of science communication. *Technical Communication Quarterly*, 26(2), 127–144. doi: 10.1080/10572252.2017.1287361
- Mehlenbacher, B., & Kampe, C. (2017). Expansive genres of play: Getting serious about game genres for the design of future learning environments. In C. R. Miller & A. R. Kelly (Eds.), *Emerging genres in new media environments* (pp. 117–133). London, England: Palgrave Macmillan.
- Miller, C. R. (1979). A humanistic rationale for technical writing. *College English*, 40(6), 610–617.
- Miller, C. R. (1984). Genre as social action. *Quarterly Journal of Speech*, 70, 151–176.
- Miller, C. R., & Kelly, A. R. (2016). Discourse genres. In A. Rocci & L. de Saussure (Eds.), *Verbal Communication* (pp. 269–286). Berlin, Germany: Mouton-De Gruyter.

- Miller, C. R., & Shepherd, D. (2004). Blogging as social action: A genre analysis of the Weblog. In L. Gurak et al. (Eds.), *Into the blogosphere: Rhetoric, community, and culture of weblogs*. University of Minnesota Libraries. Retrieved from <http://conservancy.umn.edu/handle/11299/172818>
- Miller, C. R., & Shepherd, D. (2009). Questions for genre theory from the blogosphere. In J. Giltrow & D. Stein (Eds.), *Genres in the Internet: Issues in the theory of genre* (pp. 263–290). Amsterdam, The Netherlands: John Benjamins.
- Pflugfelder, E. H. (2017). Reddit's "Explain Like I'm Five": Technical descriptions in the wild. *Technical Communication Quarterly*. Advance online publication. doi: 10.1080/10572252.2016.1257741
- Porter, J. E. (2009). Recovering delivery for digital rhetoric. *Computers and Composition*, 26(4), 207–224.
- Qiu, Y. (2016). The openness of Open Application Programming Interfaces. *Information, Communication & Society*, 20, 1720–1736.
- Randall, N. (2017). Source as Paratext: Videogame adaptations and the question of fidelity. In C. R. Miller & A. R. Kelly (Eds.), *Emerging genres in new media environments* (pp. 171–185). London, England: Palgrave Macmillan.
- Read, S. (2016). The net work genre function. *Journal of Business and Technical Communication*, 30(4), 419–450.
- Reiff, M. J., & Bawarshi, A. (Eds.). (2016). *Genre and the performance of publics*. Boulder, CO: University Press of Colorado Press.
- RobLoach. (2007). ReCAPTCHA. *Drupal*. Retrieved from <https://www.drupal.org/project/recaptcha>
- Ross, D. G. (2017). The role of ethics, culture, and artistry in scientific illustration. *Technical Communication Quarterly*, 26(2), 145–172. doi: 10.1080/10572252.2017.1287376
- Russell, D. R. (1997). Rethinking genre in school and society: An activity theory analysis. *Written Communication*, 14(4), 504–554.
- Sánchez, F. (2016). The roles of technical communication researchers in design scholarship. *Journal of Technical Writing and Communication*, 47(3), 359–391.
- Sample, M. (2013). Criminal code: Procedural logic and rhetorical excess in videogames. *Digital Humanities Quarterly*, 7(1). Retrieved from <http://www.digitalhumanities.org/dhq/vol/7/1/000153/000153.html>
- Sapienza, F. (2002). Does being technical matter? XML, single source, and technical communication. *Journal of Technical Writing and Communication*, 32(2), 155–170.
- Schryer, C. F. (1993). Records as genre. *Written communication*, 10(2), 200–234.
- Schryer, C. F. (1994). The lab vs. the clinic: Sites of competing genres. In A. Freedman & P. Medway (Eds.), *Genre and the new rhetoric* (pp. 105–124). London, England: Taylor & Francis.
- Schryer, C. F. (1999). Genre time/space: Chronotopic strategies in the experimental article. *A Journal of Composition Theory*, 19, 81–89.
- Shah, R. C., & Kesan, J. P. (2008). Setting online policy with software defaults. *Information, Communication & Society*, 11(7), 989–1007.
- Shepherd, D. (2016). *Building relationships: Online dating and the new logics of Internet culture*. New York, NY: Lexington Books.

- Sherlock, L. (2009). Genre, activity, and collaborative work and play in World of Warcraft: Places and problems of open systems in online gaming. *Journal of Business and Technical Communication*, 23(3), 263–293.
- Spinuzzi, C. (2001). Software development as mediated activity: Applying three analytical frameworks for studying compound mediation. In M. J. Northrop, & S. Tilley (Eds.), *SIGDOC '01: Proceedings of the 19th Annual International Conference on Computer Documentation* (pp. 58–67). New York: ACM Digital Library.
- Spinuzzi, C. (2002). Towards a hermeneutic understanding of programming languages. *Currents in Electronic Literacy*, 6. Retrieved from <http://currents.dwrl.utexas.edu/spring02/spinuzzi.html>
- Spinuzzi, C. (2003a). Compound mediation in software development: Using genre ecologies to study textual artifacts. In C. Bazerman & D. R. Russell (Eds.), *Writing selves/ Writing societies: Research from activity perspectives*. Fort Collins, CO: WAC Clearinghouse. Retrieved from http://wac.colostate.edu/books/selves_societies/spinuzzi/
- Spinuzzi, C. (2003b). *Tracing genres through organizations: A sociocultural approach to information design*. Cambridge, MA: MIT Press.
- Spinuzzi, C. (2013). *Topsight: A guide to studying, diagnosing, and fixing information flow in organizations*. Seattle, WA: CreateSpace Independent Publishing Platform.
- Stolley, K. (2015). MVC, materiality, and the magus: The rhetoric of source-level production. In J. Ridolfo & W. Hart-Davidson (Eds.), *Rhetoric and the digital humanities* (pp. 264–276). Chicago, IL: University of Chicago Press.
- Vee, A. (2013). Understanding computer programming as a literacy. *Literacy in Composition Studies*, 1(2). Retrieved from <http://licsjournal.org/OJS/index.php/LiCS/article/view/24>
- Wardrip-Fruin, N. (2009). *Expressive processing: Digital fictions, computer games, and software studies*. Cambridge, MA: MIT Press.
- Wickman, C. (2016). “Learning to share your science”: The scientific notebook textual object and dynamic rhetorical space. In A. G. Gross & J. Buehl (Eds.), *Science and the Internet: Communicating knowledge in a digital age* (pp. 11–32). Amityville, NY: Baywood Press.
- Wundo. (2004). CAPTCHA. *Drupal*. Retrieved from <https://www.drupal.org/project/captcha>
- Yates, J., & Orlikowski, W. J. (1992). Genres of organizational communication: A structural approach to studying communication and media. *Academy of Management Review*, 17(2), 299–326.
- Zappen, J. P. (2005). Digital rhetoric: Toward an integrated theory. *Technical Communication Quarterly*, 14(3), 319–325.

Author Biographies

Kevin Brock is an assistant professor in the Department of English Language and Literature at the University of South Carolina, USA. His research interests focus on the rhetorical nature of software and the practices surrounding its development.

Ashley Rose Mehlenbacher (formerly Kelly), PhD, is an assistant professor in English Language and Literature at the University of Waterloo, Canada. She is the coeditor, with Carolyn R. Miller, of *Emerging Genres in New Media Environments* (Palgrave Macmillan).