The 'FizzBuzz' Programming Test: A Case-Based Exploration of Rhetorical Style in Code

Kevin Brock
University of South Carolina

Abstract

While software developers have long discussed concerns of style in regards to writing code, scholars of computation would benefit from a rhetorical approach to style, an approach that links style to substance and sees style as situated and audience-specific. However, every code text is informed by stylistic decisions that impact how the text is interpreted and understood. In this essay, several stylistic variations of code written for the 'FizzBuzz' hiring test are examined in order to demonstrate the significance of stylistic choice in code composition. The range of approaches coders might take to communicate a preferred method of accomplishing a given task in code indicates that rhetorical style performs an important role in how code is accessed and comprehended by human and nonhuman audiences alike. Accordingly, software critics need to attend more closely to the ways that coders employ style in order to induce particular types of rhetorical action through their code texts and practices.

Introduction

For the last several decades, rhetoric has increasingly been understood to play a significant role in communication occurring not just in discursive speech and writing acts but across numerous modes of meaning-making, including image, color, gesture, spatial arrangement, aurality, and procedure. For rhetoricians, recognizing that meaning is created and communicated across these modes serves as a beginning point to understand how each of these modes, and the media through which they are communicated, affords particular approaches to constructing and representing persuasive arguments for various audiences. This question is commonly understood to be one focused squarely on *style*.

Rhetorical style is often described as an ornamentation or clarification of substantive argument, those ideas initially developed through invention.[1. Aristotle, *On Rhetoric: A Theory of Civic Discourse*, trans. George Kennedy (Oxford: Oxford UP, 1991), III.ii.1-3.] However, style serves a greater purpose as the articulation and performance of the values underlying one's argument. As Barry Brummett argues, 'Style is a complex system of actions, objects, and behaviors that is used to form messages that announce who we are, who we want to be, and who we want to be considered akin to.'[2. Barry Brummett, *A Rhetoric of Style* (Carbondale: Southern Illinois UP, 2008), xi.] Similarly, Chris Holcomb and M. Jimmie Killingsworth argue that stylistic variants result in messages that may appear similar but suggest distinct ways of understanding a given argument and the world it constructs.[3. Chris Holcomb and M. Jimmie Killingsworth, *Performing Prose: The Study and Practice of Style in Composition* (Carbondale: Southern Illinois UP, 2010), 3-4.] Rhetoricians since antiquity have often suggested that among style's greatest strengths relates to disguising one's stylistic decisions in order to pursue more effectively one's goals: 'authors should compose without being noticed and should speak not artificially but naturally [… Concealing stylistic decisions] is well done if one composes by choosing words from ordinary language.'[4. Aristotle, *On Rhetoric*, III.ii.4.] While calling attention to, or avoiding calling attention to, one's stylistic decisions can work in favor of some ends compared to others, style does not have to promote an agonistic or deceptive relationship between rhetor and audience. Style, described more inclusively, is how we attempt to construct effectively the artistic presentation of an argument in order to achieve a particular end.

Style as a term used among computer programmers is conventionally either (1) addressed as a set of universally applicable qualities (e.g., conciseness, readability/clarity, lack of repetition) that may not necessarily reflect the interests or values of specific communities or (2) ignored altogether as a relevant concept in discussions about 'best' or preferred programming practices. This is not to suggest that style is omitted from decision-making about how to write code—only that such decisions are not always made consciously or explicitly by developers communicating with one another through code. Here is where a rhetorical approach to style is useful: it provides a framework for understanding style as situated in particular communities, wielded toward particular ends, and crafted for particular audiences.

In this essay, I explore how rhetorical style relates to software development in order to demonstrate that writing code is a rhetorical practice of meaning-making and worthy of note as such. Just as Matthew Fuller has argued that, '[b]ecause free and open source software opens up the process of writing software in certain ways its [sic] also opens up the process of talking and thinking about it,'[5. Matthew Fuller, 'Introduction, the Stuff of Software,' in *Software Studies \ A Lexicon*, ed. Matthew Fuller (Cambridge, MA: MIT Press, 2008), 7.] so too should we investigate how processes of writing software suggest certain ways of talking about and thinking about communicating *through those acts of writing software code*. For software studies scholars, attending to rhetoric's understanding of style is particularly significant because it helps us understand how persuasive decisions have potentially major impacts on how algorithms are imagined, programs are created, and social events are impacted through the use thereof.

As noted by rhetorician Kenneth Burke, '[w]herever there is persuasion, there is rhetoric. And wherever there is "meaning," there is "persuasion."'[6. Kenneth Burke, *A Rhetoric of Motives* (Berkeley: U of California Press, 1969), 172.] My own exploration of rhetorical meaning will occur through the examination of how code developers write, and audiences make use of and respond to, stylistic decisions relating to the construction of procedure, syntax, and arrangement of ideas. As a relevant situation for identifying and illuminating style preferences as they work toward certain rhetorical ends, I will analyze examples of code written to complete the FizzBuzz test, a small-scale hiring test to examine programming applicants' basic knowledge both of code languages and of procedural and iterative operations. Despite its small size (in most iterations of texts written for the test), FizzBuzz serves as a relatively accessible synecdoche for programming as an activity and code as a text; the approaches taken to solve the test—and the myriad, often passionate discussions surrounding any individual solution to the test—can help us see more clearly how code and rhetoric are more closely intertwined than may often be recognized.

Style and Its Role in Rhetorical Activity

Among software developers, style is commonly understood as serving a seemingly universal and instrumental end, often framed in ways that suggest particular ideologies of 'best' or 'elegant' or 'acceptable' practice within an industrial frame. For example, Brian W. Kernighan and P.J. Plauger generalize the shared values of human audiences of code when, in the preface to the first edition of their *Elements of Programming Style*, they state, '*The principles of style, however, are applicable in all languages, including assembly codes.'* [sic][7. Brian W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, 2nd ed. (New York: McGraw-Hill, 1978), xi.] The issue under debate is what 'style'—in this case, the implied 'good' style of readable code—means to diverse bodies of programmers and programming communities whose preferences for how code works, and how it reads to the community members making use of it (e.g., building on existing code, rewriting existing code, or contributing new code). Kernighan and Rob Pike begin moving toward a clarification of these nuances when they note,

> [W]hy worry about style? Who cares what a program looks like if it works? [… W]ell-written code is easier to read and to understand, almost surely has fewer errors, and is likely to be smaller than code that has been carelessly tossed together and never polished. […] Sloppy code is bad code—not just awkward and hard to read, but often broken.[8. Brian W. Kernighan and Rob Pike, *The Practice of Programming* (Reading, MA: Addison-Wesley, 1999), 28.]

While still generalizing, Kernighan and Pike make several important observations. First, they respond to questions about style—specifically, about distinctions often made between style and logical ability (e.g., a difference between what a computation would do conceptually and how that might be written as a procedure in code). Second, they acknowledge that many view questions of style as unimportant or incidental. Third, they argue that such perspectives ignore the impact that stylistic choices have on the mechanical *as well as* the readable quality of code—one's inability or refusal to write in a way that makes sense to some anticipated human reader may very likely also result in an inability to write in a way that a computer can interpret successfully.

This integral role of style to anticipated action is crucial for rhetoricians, as style has traditionally been understood as the filter or frame through which ideas are shaped for presentation to an audience. In other words, if rhetorical invention is the *what* (the discovery and development) of an argument, then, through this lens, style—in combination with delivery and arrangement—would be considered *how* that argument is constructed to be presented most successfully for the appropriate audience(s). Unfortunately, for many such a definition has long constrained style as providing only ornament or decoration on a distinct argument, as though the manner of presentation were subordinate to and separate from the content of that argument. Unfortunately, this line of thinking is in line with trends to associate style with a general set of prescriptions for practice, such as demonstrated by Kernighan and Plauger above.

However, a number of rhetorical scholars have begun attending to the complex and nuanced contributions style provides to a given persuasive act, and it is this work that I will bring to bear on questions of computer programming. Brummett, following Judith Butler and other scholars of performativity, argues that style is instead a much more substantial component of an argument: it is not decoration but the *performances* we engage in when we communicate, reflecting the collection of social values that we anticipate sharing with our audience(s).[9. Brummett, *A Rhetoric of Style*, 24-26.] Holcomb and Killingsworth propose that, while meaning is often 'somehow independent of style [… a]ny change in the manner of expression will have consequences for the meanings expressed.'[10. Holcomb and Killingsworth, *Performing Prose*, 2.] In other words, style *cannot* merely be ornament added onto an otherwise standalone argument since the means of its performance are infused with, and impart to the audience, significant meaning integral to the persuasive case being made, as similarly suggested by

Kernighan and Pike above.

Further, style offers perhaps the clearest lens through which to understand a particular rhetor and his or her persuasive intent; style has been defined by Joseph A. DeVito as 'the selection and arrangement of those linguistic features which are open to choice.'[11. Joseph A. DeVito, 'Style and Stylistics: An Attempt at Definition,' *Quarterly Journal of Speech* 53.3 (1967): 249.] That is, style is the means through which a rhetor can determine how best to approach his or her case, choosing from the options that have been discovered (i.e., 'invented') as available for *this* rhetor, for *this* audience, and for *this* particular situation or event. The capacity of stylistic decisions to influence the argument is thus potent rather than insignificant. Just as the stylistic choices provided to a rhetor prove to be powerful means at his or her disposal, so too are the choices provided to an audience in how the *reception* of a given argument will occur.

All interactions between rhetor and audience serve as what Lloyd Bitzer has called 'rhetorical situations' in which a rhetor responds to or, as others have argued since, creates a particular *exigence* that requires responsive action in order to achieve some sort of situational resolution.[12. Lloyd Bitzer, 'The Rhetorical Situation,' *Philosophy & Rhetoric* 1.1 (1968): 6. Bitzer's approach—specifically, his claim that a rhetorical situation has an objective existence before and outside of a rhetor's response to it—was challenged by Richard E. Vatz, who claims that the rhetor *creates* a situation by interpreting and translating (for his or her audience) a select combination of qualities determined to be appropriately relevant or worthy of response, while more recently, Jenny Edbauer has reframed the rhetorical situation as an ecology of experiences. For more, see Richard E. Vatz, 'The Myth of the Rhetorical Situation,' *Philosophy & Rhetoric* 6.3 (1973): 154-161; Jenny Edbauer, 'Unframing Models of Public Distribution: From Rhetorical Situation to Rhetorical Ecologies,' *Rhetoric Society Quarterly* 35.4 (2005): 5-24.] Drawing on the choices available in regards to a given rhetorical situation, a skilled rhetor offers multiple avenues of engagement with an important message, and the way that an audience responds and reacts to that message through particular types of engagement has a serious impact on the audience's decision (conscious or not) to pursue the action that the rhetor seeks to effect.

However, not all persuasive efforts are so clear-cut; viewing code as a valid means of persuasion requires recognizing that, to modify DeVito's definition above, style is not *limited* to 'linguistic features' but in fact extends across a wide variety of linguistic and non-linguistic choices that have the potential to impact a particular argument in sometimes radically different and distinct ways. As Joasia Krysa and Grzesiek Sedek have described it, 'Examining the source code of a particular program reveals information about the software in much the same way as the ingredients and set of instructions of a recipe reveals information about the dish to be prepared [… B]oth programming and cooking can express intentionality and style.'[13. Joasia Krysa and Grzesiek Sedek, 'Source Code,' in *Software Studies \ A Lexicon*, edited by Matthew Fuller (Cambridge, MA: MIT Press, 2008), 236-237.] Critical engagements with code as meaningful, rhetorical writing thus involves exploring how code suggests the intent of its developer(s) for particular code functionality and structure as well as through any expressive engagements with the interpreted or executed software it comprises.

Similarly, there has been extensive discussion for decades in regards to the rhetorical (or instrumental) character of technical writing in many forms, and this debate roughly parallels discussions in computer science over Donald Knuth's argument for 'literate programming,' an approach to software development in which code would be written to be understandable enough to a human audience so as not to require supporting documentation (e.g., comments).[14. Donald E. Knuth, *Literate Programming* (Stanford, CA: Center for the Study of Language and

Information, 1992). Knuth's argument has been echoed in different language by other well-known programmers, perhaps most notably by Yukihiro Matsumoto, 'Treating Code as an Essay,' trans. Nevin Thompson, in *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson (Sebastopol, CA: O'Reilly), 477-481.] In other words, code would be composed with a rhetorical awareness and anticipation of the human reader that would encounter and work with the code in addition to the machine programs that would compile, interpret, or execute it. Among rhetoricians, this conversation centered on the rhetorical or instrumental nature of technical communication more broadly, with the majority of scholars advocating a need to recognize the former even in communication that appears to provide 'merely' instrumental or functional aid to a reader.[15. This discussion arguably began with the publication of Carolyn R. Miller's 'A Humanistic Rationale for Technical Writing,' *College English* 40.6 (1979): 610-617. It was extended in a fascinating debate between Robert R. Johnson and Patrick Moore, among others. See: Robert R. Johnson, 'Complicating Technology: Interdisciplinary Method, the Burden of Comprehension, and the Ethical Space of the Technical Communicator,' *Technical Communication Quarterly* 7.1 (1998): 75-98; Patrick Moore, 'Myths About Instrumental Discourse: A Response to Robert R. Johnson,' *Technical Communication Quarterly* 8.2 (1999): 210-223; Robert R. Johnson, 'Johnson Responds,' *Technical Communication Quarterly* 8.2 (1999): 224-226.] As Carolyn R. Miller has argued in her initial publication on this question, 'To write, to engage in any communication, is to participate in a community; to write well is to understand the conditions of one's own participation—the concepts, values, traditions, and style which permit identification with that community and determine the success or failure of communication.'[16. Miller, 'Humanistic Rationale,' 617.] It is with such a perspective that we can most effectively see how code—as a form of specialized technical discourse—offers not only a functional account of computational activity but also a set of human-oriented arguments for persuasive ends that vary by the situation, audience, and author.

Style Demonstrated in the 'FizzBuzz' Programming Test

To begin exploring some essential considerations of rhetorical style in code, I turn to the 'FizzBuzz' test, a specific code genre that emphasizes, among other communicative and problem-solving perspectives, *style* as a central component of programming ability (alongside basic functional code literacy). It exists in a peculiar rhetorical situation: the author of a FizzBuzz document is generally an applicant for a software programming position, and the audience for this document is conventionally limited to a small body of employees tasked with hiring for that position. However, FizzBuzz is *so* widely used as a rudimentary level metric of employee potential that it is well-known by most professional programmers and is commonly employed in a secondary rhetorical situation as a thought experiment for stylistic practice; as I explore below, conversations about FizzBuzz frequently trigger debates about how solutions to the test 'should' be written. Not surprisingly, these debates ultimately focus on, although rarely mention explicitly, the stylistic choices infused into individual attempts at writing effective FizzBuzz texts for specific employer audiences and with particular goals in mind for what their FizzBuzz compositions demonstrate.

While FizzBuzz is popular, it is hardly the only—or even the first—test used to determine basic programming ability or preferences in coding style. Gayle Laakman McDowell has authored books and columns on technical interviews as effectively objective demonstrations of technical skill—an approach that roughly echoes the argument for 'instrumentality' in code texts and development processes[17. Gayle Laakmann McDowell, *Cracking the Coding Interview:*

150 Programming Questions and Solutions (Palo Alto, CA: CareerCup, 2013). See also: Gayle Laakmann McDowell, 'Why Female Programmers Should Love Technical Interviews,' November 24, 2012, accessed August 31, 2015, http://women2.com/2012/11/24/why-female-programmers-should-love-technical-interviews/], although some critics, like Heidy Khlaaf, have noted that such a perspective obscures certain ways of knowing and doing relevant programming activities.[18. Heidy Khlaaf, 'Cultural Ramifications of Technical Interviews,' *Model View Culture* 23 (2015), accessed August 31, 2015, https://modelviewculture.com/pieces/cultural-ramifications-of-technical-interviews] Similarly, scholars such as Wendy Hui Kyong Chun and Alexander R. Galloway have each argued that code languages and software programming activities in general possess and promote certain ideologies while simultaneously obscuring them in a seemingly objective (and instrumental) frame.[19. See: Wendy Hui Kyong Chun, *Programmed Visions: Software and Memory* (Cambridge, MA: MIT Press, 2011); Alexander R. Galloway, 'Language Wants to be Overlooked: On Software and Ideology,' *Journal of Visual Culture* 5 (2006): 315-331.] For interview- or test-related code—which is usually both relatively compact in nature (as an extemporaneous exercise) and incredibly significant in how it 'speaks for' the author, since the code serves as a potential gateway through which one is given access to, or prevented access from, a given employment institution and the broad community of professional programmers—these obscured ideologies, and the means by which they are presented, interpreted, and otherwise framed go hand-in-hand with concerns of rhetorical style. These concerns are often verbalized as questions of elegance, computational efficiency, preference for (or against) idiomatic expressions, and so on[20. For a small selection of discussions regarding individuals' preference for particular styles and paradigms of programming, see: Kernighan and Pike, *The Practice of Programming*, 10-17; Kernighan and

Plauger, *Elements of Programming Style*, 1-7; Jon Bentley, 'The Most Beautiful Code I Never Wrote,' in *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson (Sebastopol, CA: O'Reilly, 2007), 29-40; Matsumoto, 'Treating Code,' 478-479.]; in each case, the role of such concerns in the evaluation of code reveals important information about the author of the code and the employer interpreting potential meaning in the author's efforts.

The FizzBuzz test essentially asks job candidates for programming positions to code a program (usually in a specific programming language) that will count from one to one hundred and perform the following operations:

1. Print to the screen each number, unless...
2. If a number is a multiple of three, print '`Fizz`' instead of the number.
3. If a number is a multiple of five, print '`Buzz`' instead of the number.
4. If a number is a multiple of both three and five, i.e. a multiple of fifteen, print '`FizzBuzz`' instead of the number.

The test is rarely employed to see how 'perfectly' a programmer might achieve this outcome, since most languages afford multiple approaches to completing the test scenario in a way that would satisfy both human and computer audiences. As Barry Brown explains code practices more generally, 'Programming languages thus sit in an unusual and interesting place – designed for human reading and use, but bound by what is computationally possible.'[21. Barry Brown, "The Next Line': Understanding Programmers' Work,' *TeamEthno-Online* 2 (2006): 30, accessed August 31, 2015, http://archive.cs.st-andrews.ac.uk/STSE-Handbook/Other/Team %20Ethno/TeamEthno.html] That is, writing code is exercising tensions between human and machine expectations of what that code is capable of doing. The goal of the test is to help the employer learn about *how* that programmer approaches solving the FizzBuzz problem through his or her employment of one or more programming languages—in other words, how that applicant makes effective use of rhetorical style in his or her code.

In terms of basic code principles, there are several different ways that this problem can be solved, with a common approach—conventionally described as 'elegant'—that makes use of a *loop*, a task performing iteratively and recursively the same set of operations against each member of a given sequence.[22. Wilfried Hou Je Bek, 'Loop,' in *Software Studies \ A Lexicon*, edited by Matthew Fuller (Cambridge, MA: MIT Press, 2008), 181.] In the case of FizzBuzz, this looped sequence consists of whole numbers from one to one hundred. Definitions of elegance are plentiful among programmers. Jack Dongarra and Piotr Luszczek note that, '[A]t the software level, there is a continuous tension between performance and portability on the one hand, and understandability of the underlying code.'[23. Jack Dongarra and Piotr Luszczek, 'How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination,' in *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson (Sebastopol, CA: O'Reilly, 2007), 229.] Adam Kolawa defines elegance as code that 'accurately and efficiently perform[s] the task that it was designed to complete, it such a way that there are no ambiguities as to how it will behave.'[24. Adam Kolawa, 'The Long-term Benefits of Beautiful Design,' in *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson (Sebastopol, CA: O'Reilly, 2007), 253.] Yukihiro Matsumoto uses the term beauty instead of elegance: 'Unreadable code will reduce most people's productivity significantly. On the other hand, easily understandable code will increase it. And we see beauty in such code.'[25. Matsumoto, 'Treating Code as an Essay,' 478.] In all of these cases, there is a recognition that the program's style incorporates conciseness and clarity of purpose, although how those qualities manifest in code are valued differently by these parties.

While many discourse communities prize 'elegance' in an amorphous sense, there is a specific (similar, if not entirely identical) outcome to the looping procedure *regardless* of its implementation in code. However, procedurally there exist distinctions—some trivial, some significant—between different types of loops and how they execute the relevant computation. Even if we focus on one specific loop, there remain stylistic approaches to executing that loop and providing the anticipated outcome expression. Such a variety of possible approaches to writing—albeit in other genres, if not in code—has long been known to rhetoricians: Erasmus' sixteenth-century text on *copia* outlines exercises for improving one's stylistic abilities by writing and rewriting the same text, with nearly two hundred examples, via different tropes and strategies in response to various needs.[26. Desiderius Erasmus, *Copia: Foundations of the Abundant Style*, trans. Betty I. Knott, in *Collected Works of Erasmus: Literary and Educational Writings*, edited by Craig R. Thompson (Toronto: University of Toronto Press, 1978), 279-650.] Recently, James J. Brown, Jr. has discussed the creation of *copia*-generating software as a contemporary descendant of Erasmus' student,[27. James J. Brown, Jr., 'The Machine that Therefore I Am,' *Philosophy and Rhetoric* 47.4 (2014): 494-514.] while Cristina Lopes has explored *copia* for programming more directly by offering over thirty different approaches to writing software that calculate term frequency in a given text.[28. Cristina Videira Lopes, *Exercises in Programming Style* (Boca Raton, FL: Chapman and Hall/CRC Press, 2014).]

Differences in programmers' decision-making processes here highlights nothing so much as the performative rhetorical style of the coder *as well as* of the discursive nature of the code language chosen (that is, the stylistic preferences 'baked in' to the language by its authors). Laura

R. Micciche, addressing concerns of sentence-level grammar for more conventional forms of writing, argues that '[w]ord choice and sentence structure are an expression of the way we attend to the words of others, the way we position ourselves in relation to others […] How we put our ideas into words and comprehensible forms is a dynamic process rather than one with clear boundaries between what we say and how we say it.'[29. Laura R. Micciche, 'Making a Case for Rhetorical Grammar,' *College Composition and Communication* 55.4 (2004): 719.] As demonstrated below, we can easily replace 'word' and 'sentence' with 'operation' and 'logical function' here to extend Micciche's insights beyond conventional writing to include considerations of rhetorical style in code. Some FizzBuzz examples in PHP, Ruby, and Java help make these considerations clear.

FizzBuzz in PHP

```php
<?php
for($i=1;$i<=100;$i++) {
  if (($i%3==0) || ($i%5==0)) {
    if ($i%3==0) {
      echo "Fizz";
    }
    if ($i%5==0) {
      echo "Buzz";
    }
  } else {
    echo $i;
  }
}
?>
```

```php
<?php
for($i=1;$i<=100;$i++) {
  if ($i%15==0) {
    echo "FizzBuzz";
  } elseif ($i%3==0) {
    echo "Fizz";
  } elseif ($i%5==0) {
    echo "Buzz";
  } else {
    echo $i;
  }
}
?>
```

Table 1. Two example FizzBuzz loops in PHP[30. Code examples in Tables 1 and 2 composed by the author.]

The differences between these displayed lines of code are relatively subtle, but they play a key role in understanding how stylistic choices have a significant impact on computation. The left-side function includes a hierarchical set of conditional statements. The outer `if()` statement checks to see if `$i` is a multiple of either three or five; if so, it then checks to see which text replacement condition applies (and both are potentially applicable), and if none is valid, it prints the value of `$i`. In contrast, the right-side function sets up a single condition—`$i` as a multiple of fifteen—and then provides a series of alternative and *entirely exclusive* conditions for when that initial condition is not applicable. That is, unlike the code on the left side of Table 1, the code on the table's right side will not evaluate the condition of `$i` as a multiple of 3 and `$i` as a multiple of 5 in regards to the same iteration of `$i`.

Is either of these logically 'superior' or more 'elegant' than the other, and is this a worthwhile question to consider here? Reaching such a consensus would likely be impossible, and the criteria different individuals might use to make their case (e.g., the fewer number of lines for the right-side code vs. the non-exclusivity of the conditions in the left-side code) suggest that the 'best' approach should not be considered in terms of logical superiority but instead of what is most persuasive for a particular audience or purpose. As Wilfried Hou Je Bek observes, 'If we regard the loop as a species of tool for thinking about and dealing with problems of a certain nature, the sheer light-footedness of looping allows you to run away with the problem with more ease.'[31. Hou Je Bek, 'Loop,' 180.] The criterion of 'most persuasive,' with all its contingent possibilities, is made possible only through and because of the particular needs of a given discourse community whose members share similar perspectives on what code can and should *do* to help one achieve a specific goal or solve a problem. However, what is most persuasive or elegant, as with all other concerns related to rhetoric, is highly situated and rarely remains static for long across multiple attempts (i.e., examples within a given genre) to achieve a particular type of response.

FizzBuzz in Ruby

In comparison to the two example ways of constructing a FizzBuzz loop in PHP, the distinction among potential loop styles is a bit clearer in Ruby, as demonstrated by the two examples in Table 2:

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

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

Table 2. Two example FizzBuzz loops in Ruby

In this Ruby example, the means by which an iterative loop is called (e.g., the `for` loop vs. the `times` method) influences how the coder is constrained in dealing with it, and this stylistic influence (of distinct *type* of loop) is markedly different from the style choices made in Table 1 in relation to a single type of loop. The `times` method, which is called when initializing an object belonging to the Integer class, begins at 0 and counts up to 99, meaning that addition has to occur for each iteration of the loop via the line `i = i+1`; without that line to adjust for the 'correct' number being computed in each iteration (i.e., from 1 to 100 rather than from 0 to 99), repetitive lines of code would be necessary for the function to be expressed correctly (i.e. `if (i+1)%3==0 then`, etc.). The framing of conditional statements is otherwise interchangeable, as with the PHP examples.

So why might this distinction matter? The construction of the loop, no matter the approach employed, suggests a tremendous amount about the rhetorical aims of the author in his or her efforts to solve the FizzBuzz problem and communicate that solution to the hiring body. These aims are not limited to how the author views him- or herself as an individual programmer working on a specific program but also about how that programmer sees his or her understanding of the conventions of programming in a given discourse community (e.g., the hiring organization or, beyond the scope of FizzBuzz, the active community of contributors to a particular software project). In addition, these rhetorical aims relate to concerns about how the author understands his or her relationship with a particular computer system (e.g., OS, programming language, IDE or editing environment). For example, is it more important to plan for all contingencies in parallel (as in the left-side Ruby example), for a single 'root' condition and all its possible alternatives (the right-side PHP and Ruby examples), or for a hierarchical or prioritized approach to dealing with data (the left-side PHP example)? The answer, of course, depends upon the recognized needs and concerns of a given author, his or her human audience, and the anticipated abilities of the nonhuman machine audience that will express that code. This is not to suggest that all interpretations of a rhetorical situation are equally valid but rather to acknowledge that contingency is an integral component of any communication, and code is not exceptional in this regard.

FizzBuzz in Java Enterprise Edition

Perhaps the most clearly playful (and satirical) approach to FizzBuzz exists in the form of code written for Java Enterprise Edition ('FizzBuzz Enterprise Edition'), which adheres to stylistic conventions of large-scale Enterprise Edition interfaces and environments and which its initial author, who uploaded the code to online repository GitHub under the account name EnterpriseQualityCoding, describes as 'a no-nonsense implementation of FizzBuzz made by a serious businessman for serious business purposes.'[32. EnterpriseQualityCoding, 'FizzBuzz Enterprise Edition,' December 22, 2014, accessed August 31, 2015, http://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition] The author further situates the program and the purpose for its existence in its current form:

> Enterprise software marks a special high-grade class of software that makes careful use of relevant software architecture design principles to build particularly customizable and extensible solutions to real problems. This project is an example of how the popular FizzBuzz game might be built were it subject to the high quality standards of enterprise software.[33. EnterpriseQualityCoding, 'FizzBuzz Enterprise Edition.']

Without explicitly identifying relevant decisions as relying on concerns of style, the author is clearly aware of the role that stylistic preferences and constraints play in the construction of Enterprise software at many large corporations (i.e., those companies most likely to develop such software). Further, the author calls to attention how decisions that have nothing to do with the computational logic of the language or its operation—decisions of style—have nonetheless become paramount in regards to making a 'readable' program *as Enterprise software is assumed to be written and understood*. It is telling that these decisions seem to work directly against the goals of 'style' as described in shorthand by Kernighan and Plauger or by Kernighan and Pike; the author of 'FizzBuzz Enterprise Edition' has an extensive familiarity with Enterprise programs, and that familiarity results in an understanding of 'readability' that may seem absurd but is perfectly understandable for the constellation of Enterprise developer communities and their shared needs.

It is also noteworthy that this choice of adopting a particular style is not the same as pursuing what has been called 'obfuscated progrraming,' an approach to software development wherein software is written in the most 'dense and indecipherable' ways one can compose a body of code.[34. Nick Montfort, 'Obfuscated Code,' in *Software Studies \ A Lexicon*, ed. Matthew Fuller (Cambridge, MA: MIT Press, 2008), 193.] Despite how the program has been written, the author of 'FizzBuzz Enterprise Edition' is not intentionally making this program difficult for its intended audience to encounter. Instead, the effort made to bring the program's style in line with that of other conventional (or perhaps stereotypical) Enterprise software suggests that a more 'universal' understanding of elegance would—in this context—reflect a more obfuscatory effort. The needs of its audience do not match the needs of the audiences for the PHP and Javascript examples provided earlier, and the code reflects a recognition of this distinction.

Unfortunately, the code for this version of FizzBuzz is too long to include in its entirety here; the following excerpts are intended to emphasize some of the stylistic choices the author has made that would in a more general sense be considered 'poor' or 'wrong' or 'unreadable.'

```
package
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl;

import
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.interfaces.
parameters.FizzBuzzUpperLimitParameter;
import
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.parame
ters.DefaultFizzBuzzUpperLimitParameter;

public class Main {
        public static void main(String[] args) {
                final FizzBuzz myFizzBuzz = new FizzBuzz();
                final FizzBuzzUpperLimitParameter fizzBuzzUpperLimit = new
DefaultFizzBuzzUpperLimitParameter();

myFizzBuzz.fizzBuzz(fizzBuzzUpperLimit.ObtainUpperLimitValue());
        }
}
```

From 'FizzBuzz Enterprise Edition,' file:
src/main/java/com/seriouscompany/business/java/fizzbuzz/packagenamingpackage/impl/Main.java

```
import
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.interfaces.
factories.FizzBuzzSolutionStrategyFactory;
```

```
import com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.factories.EnterpriseGradeFizzBuzzSolutionStrategyFactory;
import com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.interfaces.strategies.FizzBuzzSolutionStrategy;

public class FizzBuzz {
        public void fizzBuzz(int nFizzBuzzUpperLimit) {
                final FizzBuzzSolutionStrategyFactory mySolutionStrategyFactory =
                        new EnterpriseGradeFizzBuzzSolutionStrategyFactory();
                final FizzBuzzSolutionStrategy mySolutionStrategy =
mySolutionStrategyFactory.createFizzBuzzSolutionStrategy();
mySolutionStrategy.runSolution(nFizzBuzzUpperLimit);
        }
}
```

From 'FizzBuzz Enterprise Edition,' file: src/main/java/com/seriouscompany/business/java/fizzbuzz/packagenamingpackage/impl/FizzBuzz.java

```
public class LoopCondition {
        public boolean evaluateLoop(int nCurrentNumber, int nTotalCount) {
                final ThreeWayIntegerComparisonResult comparisonResult = ThreeWayIntegerComparator.Compare(nCurrentNumber, nTotalCount);
                if (comparisonResult == ThreeWayIntegerComparisonResult.FirstIsLessThanSecond) {
                        return true;
                } else if (comparisonResult == ThreeWayIntegerComparisonResult.FirstEqualsSecond) {
                        return true;
                } else {
                        return false;
                }
        }
}
```

From 'FizzBuzz Enterprise Edition,' file: src/main/java/com/seriouscompany/business/java/fizzbuzz/packagenamingpackage/impl/loop/LoopCondition.java

Table 3. Excerpts from 'FizzBuzz Enterprise Edition'

These excerpts demonstrate what might be considered laughably bad programming practice (and, in fact, it is almost certain the author intended precisely this response). For example, compare the brief excerpts above from the complete FizzBuzz program written in standard Java in Table 4 below. The programming styles appear, to *some* audiences, to be starkly, obviously different from one another; the non-Enterprise version reflects stylistic preferences similar to those found in the PHP and Javascript examples discussed earlier. However, this is not to suggest that the Enterprise Java code has any inherently 'poor' or 'bad' style, or that its author is unaware of the role that style has played in the construction of that program. In fact, the GitHub repository where the code for 'FizzBuzz Enterprise Edition' is hosted currently has a dozen open pull requests and ninety open issues for discussion, many centered on how the program might be more accurately or faithfully revised to reflect conventions of Enterprise programming.[35. For example, see: Leviter, 'Method Parameters Should Be Made Final,' January 2, 2015, accessed August 31, 2015, https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/pull/172; romnempire, 'Made Tests Platform-Independent and Reduced Redundancy,' May 19, 2015, accessed August 31, 2015, https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/pull/209]

```
public class FizzBuzz {                                // Everything in Java is a class

  public static void main(String[] args) {             // Every program must have main()

    for(int i = 1; i <= 100; i++) {                    // count from 1 to 100

      if (((i % 5) == 0) && ((i % 7) == 0))            // A multiple of both?

        System.out.print("fizzbuzz");

      else if ((i % 5) == 0) System.out.print("fizz"); // else a multiple of 5?

      else if ((i % 7) == 0) System.out.print("buzz"); // else a multiple of 7?

      else System.out.print(i);                        // else just print it

      System.out.print(" ");

    }
```

```
      System.out.println();

    }

  }
  Example and comments written by David Flanagan [XXX -
  http://examples.oreilly.com/jenut/FizzBuzz.java]
```

Table 4. FizzBuzz in Java

However, just as many individuals involved in these discussions seem to misunderstand some of these conventions or the playful nature of the project altogether. What makes such misunderstandings interesting is not the contributors' adherence to the premise of corporate development as a natural preference for coding style. Rather, it is how the contributors' arguments work to create a nuanced understanding of effective persuasion through attention to reader constraints. For example, one closed issue thread deals with the following problem, posed by a would-be contributor (who titled the issue 'Poor style in interface definition'): 'Interfaces and their methods are public by default, so public interface A { public void B() } should be interface A { void B() }.'[35. d53dave and Mikkeren, 'Poor Style in Interface Definition,' December 21, 2014, accessed August 31, 2015, http://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/issues/167] Essentially, the claim posits that the program's code should adhere to 'idiomatic' constructions of Java code, which Kernighan and Pike justify by stating that 'Native speakers recognize [the idiomatic form] without study and write it correctly without a moment's thought.'[36. Kernighan and Pike, 12.] However, it is precisely this 'native speaker' bias toward the idiomatic form that leads one of the project's administrators to reject the claim: 'Since our developers are from varying cultures and have a variety of different setups, we cannot rely on defaults for any declarations. This is specified in the company coding guidelines, but I am afraid these are confidential and as such, I cannot share any more information.'[37. d53dave and Mikkeren, 'Poor Style in Interface Definition.'] While the response comment ends with a nod to the humorous nature of the program's existence, its author nonetheless is acutely aware of the potential issues with assuming a particular stylistic convention that may be 'better' for some developers but which is unhelpful for *this particular anticipated reader community*. Concerns of efficiency are diminished in relation to concerns of readability and (presumably) further development and employment among the various specific situations where a programmer may decide to make use of this application and needs to grasp its context through the lens of Enterprise development more broadly.

Arguments About FizzBuzz

While the existence of numerous approaches to composing a FizzBuzz program assists such rhetorical considerations in indicating that there is no one universally perfect or optimal means of solving problems in code more broadly (nor are definitions of 'perfect' shared universally across different communities), there are nonetheless myriad discussions that inevitably occur *about* these variations in code and which sorts of code compositions are 'better' than others—sometimes defined generally or abstractly, while at other times defined with very specific contextual variables about the parameters of superiority. Such discussions demonstrate nothing so much as the significance of rhetorical style articulated in and through code as a way to reflect specific preferences for and perspectives on models of computation and software development.

It is necessary to note that, based on the number of discussions about the FizzBuzz test, and passion or ferocity with which discussions about it occur, we cannot simply categorize the test as merely a demonstration of some fundamental programming skill, whose output can be interpreted as either entirely successful or not. While the test certainly incorporates such an evaluation in its fold, this is not the *extent* of the test—it does not reflect a phatic expression of computational communication but rather a significant and meaningful articulation of the logical approach, idiomatic preferences, and other stylistic values performed by the author through the written code text. Accordingly, discussions *about* FizzBuzz work implicitly (and, to a lesser and less frequent extent, explicitly) to deliberate on the strengths or failures of particular styles in code and the act of programming.

Jeff Atwood has suggested on his programming blog *Coding Horror* that the FizzBuzz test is 'the simplest of programs' for professional programmers to compose, but he laments what he perceives as the inability of most programmers to solve 'tiny problems' despite the relative ease of the test.[38. Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, February 26, 2007, accessed August 31, 2015, http://blog.codinghorror.com/why-cant-programmers-program/] Atwood thus essentially positions himself against the development community's diverse set of principles by which one might approach the FizzBuzz test or any other dilemma. After all, 'tiny problems,' his post implies, require trivial solutions, and one's inability to use such methods to complete the test indicate that he or she was not worthy of a programming career.[39. Atwood, 'Why Can't Programmers.. Program?'] Atwood ultimately seems to follow Imran Ghory in championing an instrumental view of code and of the FizzBuzz test as a phatic act of communication; the solution to a simple problem requires little effort and a short amount of time to achieve.[40. Imran Ghory, 'Using FizzBuzz to Find Developers who Grok Coding,' *Imran on Tech*, January 24, 2007, accessed August 31, 2015, http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/] However, Atwood hints at the potential impact of stylistics on one's engagement and understanding of the test: '[t]he mechanical part of writing and solving FizzBuzz is irrelevant.'[41. Jeff Atwood (codinghorror), February 2007, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/why-cant-programmers-program/612/14] If the test were entirely instrumental, then such concerns would have to be relevant, since they would presumably be at the center of the test's evaluative criteria. That they are not suggests that a crucial rhetorical quality—which includes, but is not limited to, style—is at play in the employment of, response to, and interpretation of the FizzBuzz test.

In over four hundred comments to Atwood's post (at the time of this publication, the ability to comment on that thread has been disabled), and in nearly one hundred and fifty comments to a follow-up post by Atwood written the next day,[42. Jeff Atwood, 'FizzBuzz: The Programmer's Stairway to Heaven,' *Coding Horror*, February 27, 2007, accessed August 31, 2015, http://blog.codinghorror.com/fizzbuzz-the-programmers-stairway-to-heaven/] numerous software developers demonstrate to one another the myriad ways one could successfully approach the test. The amount of discussion generated suggests that the test, while conceptually 'simple,' offers nonetheless deep engagement with style, although no one has claimed that all interpretations are desirable or even viable (for example, a number of commenters offered 'solutions' that left unaddressed the numbers to be printed that were not multiples of three or five). Further, significant discussion centered on social and cultural concerns surrounding the test: what sort of knowledge would be considered relevant expertise, what sort of voices could comment critically with relative impunity (or with notable resistance from others), or even which members of the community had the luxury of being reputable enough to avoid or ignore being asked to perform this sort of test or exercise in an actual interview.

One commenter on Atwood's post, using the handle AndyToo, responds with a question to Atwood's complaint: 'Does that mean that every one of those people who do program but don't use (say) recursion is a bad programmer?'[43. AndyToo, February 2007, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/why-cant-programmers-program/612/4] In other words, Atwood's critique was received by other developers as missing the point, i.e. the ways developers make use of style, whether consciously or not, comprises a varied set of nuanced approaches based on myriad contingent variables, just like those of any other rhetorical situation. Other bloggers writing posts with similar concerns about FizzBuzz have had just as many impassioned responses arguing from a variety of positions about code style (and the programming competency performed by a FizzBuzz submission), although rarely if ever was the term employed: Ghory's initial blog post garnered over nine hundred replies;[44. Ghory, 'Using FizzBuzz.'] Reginald Braithwaite received a number of responses to his own post 'Don't Overthink FizzBuzz';[45. Reginald Braithwaite, 'Don't Overthink FizzBuzz,' *Raganwald*, January 24, 2007, accessed August 31, 2015, http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html] Joe Devilla's more recent blog entry 'FizzBuzz Still Works' accumulated dozens of response posts;[46. Joey Devilla, 'FizzBuzz Still Works,' *Global Nerdy*, November 15, 2012, accessed August 31, 2015, http://www.globalnerdy.com/2012/11/15/fizzbuzz-still-works/] the entry for 'Fizz Buzz Test' on the Cunningham & Cunningham wiki has been revised 574 times as of August 2015.[47. 'Fizz Buzz Test,' *Cunningham & Cunningham*, December 23, 2014, accessed August 31, 2015, http://c2.com/cgi/wiki?FizzBuzzTest]

User David_Cook shares a similar concern to AndyToo about Atwood's apparent instrumental approach to writing code: 'Easy requirements are still often misunderstood […

w]hich is why being able to correctly code simple problems is a good test of your programming (not just coding!) abilities.'[48. David_Cook, February 2010, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/why-cant-programmers-program/612/397] For this respondent, the distinction between programming and coding is inherently one of effective communication (i.e., audience-oriented persuasion). User anon84 asks explicitly about style concerns, wondering about possible unstated parameters of the problem that might influence the composition of an appropriate solution.[49. anon84, February 2007, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/why-cant-programmers-program/612/59]

Related issues about stylistic decisions as indicators of personality and behavior (i.e., as a performance of particular values) are raised by commenters like David_Cook, who claims in another comment that, '[I]f I was hiring, I would check the “readability” and “understandability” of the solution. Tricky and cute code that is unmodifiable and decipherable might work - but how well will the developer work with others? And how likely is his/her code going to adapt to continually changing requirements?'[50. David_Cook, February 2010, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/fizzbuzz-solution-dumping-ground/1752/138] This comment implies that one's coding style illuminates how well one understands code as a communicative medium—how well it can be read by a human audience and how adaptable it is for a computer interpreter. The notion of code's persuasive quality is supported by user LeeP, who argues, 'The most powerful language I “code” in is English.'[51. LeeP, February 2010, comment on Jeff Atwood, 'Why Can't Programmers.. Program?,' *Coding Horror*, accessed August 31, 2015, http://discourse.codinghorror.com/t/why-cant-programmers-program/612/408] That programmers see connections between different forms of communication rather than totalizing distinctions is further evidence not just of code's rhetorical qualities but of programmers' recognition of the available stylistic choices they make to persuade various audiences *through* their code.

As long as coders have the ability to choose how to address a given problem or task, they will engage in a stylistic performance reflecting their values and those they believe their audience to possess. This engagement becomes especially significant if one codes in a collaborative setting where it is highly likely other coders will access, build upon, or respond to one's work. This set of considerations on the part of the coder-rhetor extends to the computer system in addition to other people: the way a coder understands a code language to work will influence and reflect that coder's actions in facilitating particular behavior and results within and through the programs he or she creates. This is not to say that any of the individuals involved will consciously recognize their employment of particular styles; however, the style(s) chosen will impact how one's code is received by its various audiences.

Conclusions

While a machine interpreter cannot weigh the appropriateness of a given statement or command, its developers influence and constrain ranges of action available to those who make use of the machine and its languages to communicate with machine and human audiences. Writing about the composition of effective code in JavaScript, Douglas Crockford notes, 'Programmers can debate endlessly on what constitutes good style. Most programmers are firmly rooted in what they're used to [… and s]ome have had profitable careers with no sense of style at all. […] It turns out that style matters in programming for the same reason that it matters in writing. It makes for better reading.'[52. Douglas Crockford, *Javascript: The Good Parts*, (Sebastopol, CA: O'Reilly Media, 2008), 95.] In other words, it is the potential for powerful performance of meaning *through style*, in fluid and dynamic—rather than prescribed—ways that makes clear the value of conscious rhetorical awareness in the composition, and study, of code.

This flexible rhetoricity of code—to simultaneously enable and restrict certain types of activities—is the essence of its stylistic potential, such as how communities of developers debate the merits of performing functions in specific ways so as to reflect paradigms not just of human philosophy but of computational efficiency and elegance. For example, Lopes' demonstration of style influencing computational activity through *copia* by exploring thirty-three different ways to calculate term frequency illustrate, there are significant distinctions in coding approaches when using (or avoiding) external libraries, writing long lines of code, using functional parameters, and so on.[53. Lopes, *Exercises*, xi-xix.] If Brummett's definition of style—'a complex system of actions, objects, and behaviors that is used to form messages that announce who we are, who we want to be, and who we want to be considered akin to'[54. Brummett, *A Rhetoric of Style*, xi.]—holds, then it is in these rhetorical qualities of code that we can see such values and behaviors enacted across humans and computer systems alike. Further, style becomes understood as a conscious set of decisions made by a rhetor (as well as an unconscious set of decisions that help define us nonetheless) about how he or she wants to engage with a given audience through the message he or she has created in code and in its resulting program expression(s).

Relatively compact, clearly situated code cases like iterations of FizzBuzz provide some helpful glimpses into how code operates rhetorically at a basic level. These cases also demonstrate how code *is understood to be rhetorically significant* by programmers and others connected to the software development industry. Very little code can be reduced to the sort of singular context in which FizzBuzz exists, but if even this 'trivial' program spurs such heated debate and intense discussion about how to program correctly (or well), then more complex and extensive programs are likely to be exponentially more significant in a rhetorical sense than FizzBuzz. This is not to suggest FizzBuzz is ultimately unimportant but instead to suggest that such rhetorical concerns exist in software code of numerous scales of functional or computational importance.

For software studies, this is a significant opportunity for acknowledging, recognizing, and understanding rhetoric as it occurs in, through, and around the development of any given software program. Rhetorical style as demonstrated in code suggests a significance in even seemingly trivial spaces, from logical ordering to syntax to indentation. For the FizzBuzz test, we can witness the value in comprehending the sheer range of approaches a rhetor may engage in through code so that he or she might impart meaning to his or her audience about the code's purpose, functionality, and the relevant values of its author. As Mark Backman has noted, 'In the curriculum of the schools rhetoric has been assigned a much reduced role when the motive has been to establish discrete disciplines […] Conversely, rhetoric has organized the entire course of study when the goal has been to bridge the gap between distinct subject matters.'[55. Mark Backman, 'Introduction: Richard McKeon and the Renaissance of Rhetoric,' in Richard McKeon, *Rhetoric: Essays in Invention and Discovery*, ed. Mark Backman (Woodbridge, CT: Ox Bow Press, 1987), xix.] Rhetoric provides software critics a useful lens through which we can recognize and make meaningful use of the varying successes and failures of attempted communication across a variety of media. For software, these persuasive attempts occur as computational processes, realized in the form of written code, to achieve some sort of action among audiences. By connecting together the seemingly disparate components of these communicative engagements with the computational logics they describe and argue for or against, we can more clearly and fully explore just how software works, how its authors argue for its working, and how we might experience it as a creative and deliberative set of texts and efforts at effecting social action through procedure.

Bibliography

Aristotle. *On Rhetoric: A Theory of Civic Discourse*. Translated by George Kennedy. Oxford: Oxford UP, 1991.

Arnold, Ken. 'Style is Substance.' In *The Best Software Writing I*, edited by Joel Spolsky, 1-6. Berkeley, CA: Apress, 2005.

Atwood, Jeff. 'FizzBuzz: The Programmer's Stairway to Heaven,' *Coding Horror.* Accessed August 31, 2015. http://blog.codinghorror.com/fizzbuzz-the-programmers-stairway-to-heaven/.

Atwood, Jeff. 'Why Can't Programmers.. Program?' *Coding Horror*. Accessed August 31, 2015. http://www.codinghorror.com/blog/2007/02/why-cant-programmers-program.html.

Backman, Mark. 'Introduction: Richard McKeon and the Renaissance of Rhetoric.' In Richard McKeon, *Rhetoric: Essays in Invention and Discovery*, edited by Mark Backman, vii-xxxii. Woodbridge, CT: Ox Bow Press, 1987.

Bentley, Jon. 'The Most Beautiful Code I Never Wrote.' In *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson, 29-40. Sebastopol, CA: O'Reilly, 2007.

Bitzer, Lloyd. 'The Rhetorical Situation.' *Philosophy & Rhetoric* 1.1 (1968): 1-14.

Braithwaite, Reginald. 'Don't Overthink FizzBuzz.' *Raganwald*. Accessed August 31, 2015. http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html.

Brown, Barry. ''The Next Line': Understanding Programmers' Work.' *TeamEthno-Online* 2 (2006): 25-33. Accessed August 31, 2015. http://archive.cs.st-andrews.ac.uk/STSE-Handbook/Other/Team%20Ethno/TeamEthno.html.

Brown, James J., Jr. 'The Machine that Therefore I Am.' *Philosophy and Rhetoric* 47.4 (2014): 494-514.

Brummett, Barry. *A Rhetoric of Style.* Carbondale: Southern Illinois UP, 2008.

Burke, Kenneth. *A Rhetoric of Motives*. Berkeley: U of California Press, 1969.

Chun, Wendy Hui Kyong. *Programmed Visions: Software and Memory*. Cambridge, MA: MIT Press, 2011.

Crockford, Douglas. *JavaScript: The Good Parts.* Sebastopol, CA: O'Reilly Media, 2008.

d53dave and Mikkeren. 'Poor Style in Interface Definition.' *GitHub*. Accessed August 31, 2015. http://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/issues/167/.

Devilla, Joey. 'FizzBuzz Still Works.' *Global Nerdy*. Accessed August 31, 2015, http://www.globalnerdy.com/2012/11/15/fizzbuzz-still-works/.

DeVito, Joseph A. 'Style and Stylistics: An Attempt at Definition.' *Quarterly Journal of Speech* 53.3 (1967): 248-255.

Dongarra, Jack and Piotr Luszczek. 'How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination.' In *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson, 229-252. Sebastopol, CA: O'Reilly, 2007.

Edbauer, Jenny. 'Unframing Models of Public Distribution: From Rhetorical Situation to Rhetorical Ecologies.' *Rhetoric Society Quarterly* 35.4 (2005): 5-24.

EnterpriseQualityCoding. 'FizzBuzz Enterprise Edition.' *GitHub*. Accessed August 31, 2015. http://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/.

Erasmus, Desiderius. *Copia: Foundations of the Abundant Style.* Trans. Betty I. Knott. In *Collected Works of Erasmus: Literary and Educational Writings*, edited by Craig R. Thompson, 279-650. Toronto: University of Toronto Press, 1978.

'Fizz Buzz Test.' *Cunningham & Cunningham*. Accessed August 31, 2015. http://c2.com/cgi/wiki?FizzBuzzTest.

Fuller, Matthew. 'Introduction, the Stuff of Software.' *Software Studies \ A Lexicon*, edited by Matthew Fuller, 1-13. Cambridge, MA: MIT Press, 2008.

Galloway, Alexander R. 'Language Wants to be Overlooked: On Software and Ideology.' *Journal of Visual Culture* 5 (2006): 315-331.

Ghory, Imran. 'Using FizzBuzz to Find Developers Who Grok Coding.' *Imran on Tech*. Accessed August 31, 2015. http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/.

Holcomb, Chris and M. Jamie Killingsworth. *Performing Prose: The Study and Practice of Style in Composition*. Carbondale: Southern Illinois UP, 2010.

Hou Je Bek, Wilfried. 'Loop.' In *Software Studies \ A Lexicon*, edited by Matthew Fuller, 179-183. Cambridge, MA: MIT Press, 2008.

Johnson, Robert R. 'Complicating Technology: Interdisciplinary Method, the Burden of Comprehension, and the Ethical Space of the Technical Communicator.' *Technical Communication Quarterly* 7.1 (1998): 75-98.

Johnson, Robert R. 'Johnson Responds.' *Technical Communication Quarterly* 8.2 (1999): 224-226.

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

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

Khlaaf, Heidy. 'Cultural Ramifications of Technical Interviews.' *Model View Culture* 23 (2015).

Accessed August 31, 2015. https://modelviewculture.com/pieces/cultural-ramifications-of-technical-interviews/.

Knuth, Donald E. *Literate Programming.* Stanford, CA: Center for the Study of Language and Information, 1992.

Kolawa, Adam. 'The Long-term Benefits of Beautiful Design.' In *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson, 253-266. Sebastopol, CA: O'Reilly, 2007.

Krysa, Joasia and Grzesiek Sedek. 'Source Code.' In *Software Studies \ A Lexicon*, edited by Matthew Fuller, 236-243. Cam 'bridge, MA: MIT Press, 2008.

Leviter. 'Method Parameters Should Be Made Final.' *GitHub*. Accessed August 31, 2015. https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/pull/172

Lopes, Cristina Videira. *Exercises in Programming Style*. Boca Raton, FL: Chapman and Hall/CRC Press, 2014.

Matsumoto, Yukihiro. 'Treating Code as an Essay.' Trans. Nevin Thompson. In *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson, 477-481. Sebastopol, CA: O'Reilly, 2007.

McDowell, Gayle Laakmann. *Cracking the Coding Interview: 150 Programming Questions and Solutions.* Palo Alto, CA: CareerCup, 2013.

Micciche, Laura R. 'Making a Case for Rhetorical Grammar.' *College Composition and Communication* 55.4 (2004): 716-737.

Miller, Carolyn R. 'A Humanistic Rationale for Technical Writing.' *College English* 40.6 (1979): 610-617.

Montfort, Nick. 'Obfuscated Code.' In *Software Studies \ A Lexicon*, edited by Matthew Fuller, 193-199. Cambridge, MA: MIT Press, 2008.

Moore, Patrick. 'Myths About Instrumental Discourse: A Response to Robert R. Johnson.' *Technical Communication Quarterly* 8.2 (1999): 210-223.

romnempire. 'Made Tests Platform-Independent and Reduced Redundancy.' *GitHub*. Accessed August 31, 2015. https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition/pull/209

Vatz, Richard E. 'The Myth of the Rhetorical Situation.' *Philosophy & Rhetoric* 6.3 (1973): 154-161.