Available online at www.sciencedirect.com

Computers and Composition 23 (2006) 430–443

Computers and Composition

www.elsevier.com/locate/compcom

Coding with power: Toward a rhetoric of computer coding and composition

Robert E. Cummings

Columbus State University, Columbus, GA 31907, United States

Abstract

This article explores the connection between computer programming (coding) and traditional composition. It first looks at how a discussion about adopting the open source community's copyleft publishing model suggests a deeper parallel between coding and composition than has been previously acknowledged. By repositioning the rhetorical triangle as a coding triangle, the article argues that the act of writing programs for a machine informs the process of constructing an audience, in traditional composition. To better inform the act of how a traditional writer invokes an audience, the article summarizes how Walter J. Ong has characterized this process. The article then examines how Claudia Herbst's portrayal of the cultural power of computer code raises questions as to how computer users similarly invoke an author. It also briefly considers how computer programming texts have characterized coding as an act of writing. Next the parallels between coding and composition and their treatment thus far in composition literature and new media theory are considered. The article considers Alfred Kern's work in employing BASIC programming to teach grammar and composition and then offers suggestions for thinking of new ways that a knowledge of coding can inform the teaching of writing. This article concludes with guidelines for writing teachers who wish to incorporate computer coding into their curriculum.

Keywords: Coding; Composition; Open source; Rhetorical triangle; Audience; Teaching; XML

The 2002 Computers and Writing Conference hosted a session that asked this question: Would the terms of open source software licensing be desirable for traditional academic publishing? (Computers, 2002) The open source model, or "copyleft," differs substantially from the standard copyright procedure that governs almost all academic publishing. Under the terms of U.S. copyright, the government grants the creator of a document the right to republish that work. In the world of academic publishing, copyright is usually sold (more likely assigned to the publisher for free) upon acceptance in an academic journal or press. Unlike the commercial copyright world where millions of dollars are at stake for an author, such as John Grisham or Stephen King, academic writers anticipate little to no monetary return for their work. While

* *Email address:* cummings_robert1@colstate.edu.

8755-4615/$ – see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.compcom.2006.08.002

there are some notable exceptions involving textbooks that hold market share for students, most academic publishing is essentially a money-less proposition. Neither party in the "sale" of academic copyright, neither the scholar nor the academic press, is making much, if any, money. And it is this lack of money that threatens the very existence of academic publishing as university presses are beginning to restrict the total number of publications. Thus, many of the participants at this session came ready to hear of a new publishing model.

Composition scholars have begun to look to the software publishing world for a solution. In the last decade, a revolution has occurred in the way people write and publish software code. Instead of holding copyright over their finished product, coders in the open source world have developed copyleft (Stallman, 2002). Copyleft employs the legal protections of a copyrighted document, but instead of demanding payment for redistribution of copyrighted material, copyleft allows the program to be freely distributed as long as no recipient of that copylefted program (or subsequent program which contains the original copylefted program) charges for the program. Thus, no one can take the free program and begin charging for it, for as long as it contains the copylefted program, reselling the program is illegal under copyright law. For the world of electronic publishing, where information can be exchanged, reproduced, and distributed with little or no expense, a copyleft policy often needs not account for a way to help publishers recoup the printing and marketing expenses associated with traditional print documents.

But copyleft has still another quality which makes for an apt fit with the academic world: It promotes software development by encouraging its evolution. Anyone is free to take a copyleft program, modify it for his or her own purposes, and then redistribute it as long as he or she does not charge for it. Since no one needs to obtain permission from an original author before redeploying that code, the state of computer programming improves as a whole as open source programmers freely take and adapt a growing library of code at their disposal. Meanwhile, advocates for adopting the open source publishing method in the academic world find themselves in a publishing market so bereft of funding that many academic presses have closed, leading to a crisis for scholars whose tenure and promotion committees are fixated upon the imprimatur of book publication as the *sine qua non* of academic advancement (MLA, 2002). Surely, these composition scholars reason, with access to traditional academic publishing stifled under the profit-motive system, scholars should adopt the open source model of publication. This approach could free both writers and publishers from the profit motive, with a collateral benefit of freeing readers to expand on published ideas without obtaining copyright permission. Though there are many unanswered questions about this switch (i.e., Would authors who do receive substantial compensation for their textbooks join the movement? How would peer-reviewed journals and tenure and promotion committees evaluate work that cannot be assigned to just one author?), it would seem that the composition world is turning to the world of computer programming for a model to solve its problems.

What I would like to argue here, regardless of the outcome of this debate, is that the relationship between the coding community and the composition community as embodied in the open source publishing question is not accidental. Though their linkage is relatively unexamined and under-appreciated, both pursuits are inextricably joined by the fact that they center around the act of writers writing. Both types of writers—writers of code and writers of text—write for vastly different audiences and with what would seem to be vastly different products, but this paper will show that the underlying model of composition holds true for both communities. What writers know about the business of writing can inform and guide the coding community, and coders have learned lessons about writing that readily apply to traditional writing formats. The knowledge that these two communities might share is tremendous, and herein I would like to begin by examining and amplifying the remarkable parallels between the acts of writing for a machine and writing for humans. While Paul LeBlanc's publication of *Writing Teachers Writing Software* in 1993 has served as a landmark for this perspective on writing, and further attention was garnered in a 1999 special issue of *Computers and Composition* entitled "From Codex to Code: Programming and the Composition Classroom," informing scholars in the composition community about the benefits of examining coding as a writing practice has remained difficult. There are at least two important sources beyond the literature of the composition community to consider: new media theorists, such as Friedrich Kittler and Claudia Herbst, and the authors of practical programming texts.

As Herbst (2002) has pointed out, beyond the fact that writing computer code substitutes a machine for a human reader, code is a form of text that lacks a narrator. Herbst agrees with the central assertion of this essay in writing that "[p]rogramming languages, code, represent not only new language forms but, more important, a new form of text" (p. 1). Herbst argues that code is important to non-programmers as well as programmers not only because code affects the programming that we may use but also because it "informs technology; technology in turn informs culture" (p. 1). Herbst finds in code a new form of text with tremendous cultural influence and political power but with only a select few with access to read it:

> Previously, text concerned with power and its distribution has been not only visible by means of symbol and representation but also widely accessible, such as has been the case with judicial and religious texts. Code on the other hand, while it reaches the masses by way of technology, is largely invisible to the general population. Code is a non-public, if not stealth, form of text. [. . .] The nature of the text is the nature of its power. Among code's most defining characteristics are its inaccessibility and covert nature. The authors of code are numerous and largely anonymous. (p. 3)

While Herbst is correct in pointing to the anonymity of coders, she elides the difference between machine and human as reader of code. This distinction is important, for as will be examined herein, the programmer writes for the machine first, much as the human writer might write for an editor. But while student programmers might become so focused on the machine's reaction to the code as to lose sight of the end user, the human reaction to the program, the hallmark of the professional coder is the successful management of both audiences. Successful use of code to teach writing will create an awareness of both human and machine readers. Still, Herbst is insightful in reminding us that many readers have coding knowledge. How does the programmer write for an audience that can both operate the program as user and read the intent of the code that creates that program?

At its heart, the comparison between writing and coding is instructive to both pursuits because the work product of both groups holds meta-cognitive sway over the thinking processes that create text. That is to say that both writer and coder are moved by the manner in which their text, once written, impacts the thinking process that composed it. Writers often get hooked on writing because of its ability to impact the thoughts that create it, often in a surprising way. Many writers find in their work a therapeutic release of mental and spiritual energy, and their relationship with their finished product is characterized by the aesthetic concerns of an artist. Indeed it is the open-ended and unfamiliar nature of writing which keeps most writers coming back for more. If our model of comparing traditional writing to coding holds firm, then we should expect that programmers' descriptions of the coding process should mimic the traditional writers' descriptions of composition.

And in fact they often do. The quality of refining an idea is one that coders often cite as a compelling and rewarding aspect of their work. Their texts also speak back to their processes of composition, as revealed in the reflections on coding in some standard texts. Harold Abelson, Gerald J. Sussman, and Julie Sussman's *Structure and Interpretation of Computer Programs* (1996) begins by stating, "A programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes" (p. 4). Thus, programmers, like writers, often reflect that the mental process of writing code teaches the coder as he or she progresses: The act of applying the logic of a programming language to a problem refines that problem, positions it in a new light, and reveals the biases or faults of the thinking that first framed the issue as a problem. Coders also describe their process as an artistic one. Donald Knuth, author of one of the most respected works in Computer Science, *The Art of Computer Programming* (1997), states it as follows: "The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music" (p. v).

In fact, *The Pragmatic Programmer* (2002) pushed the relationship between human language and computer language when it boldly instructs its readers to "treat English as just another programming language" (p. 248). Then perhaps the next step is the integration of programming languages with human languages. In fact, some documents already do exactly this by providing computer code alongside a human-readable narrative; the text of the document is human readable while the code can be processed by a machine (Ramsay, 2002). Thus, code is already employed in ways that blur the lines of authors' assumptions about the audience's ability to read code, much as Herbst has foreseen. With these parallels between coding and composing, one would anticipate that many composition teachers would want to inform their pedagogy with a coding awareness.

Relatively scant attention has been paid to this perspective by mainstream composition approaches, and, further, whenever attention has been focused on this intersection of writing within two distinct fields, it has suffered from a lack of a unified vision. Therefore, this essay will investigate the rhetorical triangle as a common touchstone for both coding and composing. It will also examine some of the reasons why writing teachers have been reluctant to re-envision their writing classrooms in terms of what coding can offer.

But before we can appreciate why the lessons of coding have been slow to make their way into the classroom, it is necessary to briefly map out the specifics of this comparison. Let's quickly look at each writing paradigm (while temporarily abstaining from the multitude of theoretical divergences that the model has spawned) in order to see the similarities between writing and coding. In keeping with the spirit of open source programming, this essay will attempt to take an existing fundamental rhetorical concept—the rhetorical triangle—alter it, and then see if the result adds understanding for both pursuits.

1. Coding and composing with two triangles

In the conventional writing model, the writer seeks to reach an audience through text. The writer composes that text in any number of ways (i.e., using pen and paper, a word processor, chalk and slate, spray paint and wall, etc.), but the assumption is that the audience can only receive that message by reading the text: The author will not be able to mediate the reader's interpretation of his or her message once the text is turned over to the readers. The author's ability to successfully communicate his or her message depends on many things, not the least of which will be a shared language between writer and reader and the writer's skillful employment of that language (brushing aside centuries of rhetorical, structuralist, deconstructionist, and reader-response theoretical criticism for the moment). Successful communication of the writer's message will also depend on a reader who is actively engaged in seeking out the writer's meaning in the text even though the reader is thwarted by any number of obstacles on the way to that message, not the least of which will be the notion that every reader creates his or her own subjective meaning from text, independent of the author's intentions. Similarly, the writer can never be sure that the text he or she creates is a faithful and accurate representation of the ideas in his or her head: The written product diverges from the mental impulse that inspired it. Still the writer bravely presses onward and, after reading the written product, comes to realize that although the text is different from what he or she imagined, the text is now in control. At some point the writer surrenders control of the document by publishing it, mailing it, posting it, filing it, destroying it, or turning it over to someone else, depending on the writer's acquiescence to readers' demands.

Let's think about the computer programmer, or coder, for a moment. The coder seeks to reach an audience through his or her writing too, only the coder's first audience is a machine (one might argue that the coder's audience is the computer user instead—an important distinction that will be answered herein), and the coder's language is computer code. Like the writer, the coder can use any number of tools to create a program. While most contemporary coders might use one of a variety of all-purpose editor programs, such as Emacs, XEmacs, or vi, to produce text, during the mainframe days of programming, most coders preferred to work with pencil and paper (Baron, 1999). But regardless of the coder's tool, the machine will be the receiver of the text; like the writer who produces a text that he or she cannot further mediate, so too does the coder surrender the written text to an audience. The interpretation of that text by the machine is well beyond the coder's control once he or she "publishes" it. Like the traditional writer, the coder must also write in the same language as his or her audience: No coder would intentionally write a program in perl and then attempt to run the program on a machine that didn't have the perl language loaded on it. Again, like the traditional writer, the coder hopes that the idea in his or her head will be clearly interpreted by his or her audience. But while there may be many obstacles in the writer's mind as to how the machine will interpret the meaning of the text, there are none for the machine: The machine doesn't read text—it can only process it according to the rules of the processing language. Similarly, the code is out of the programmer's hands, and the machine is in control of determining its meaning. The coder is told—often in a fraction of a second—through the machine's response whether or not the text is interpreted the way he or she predicted.

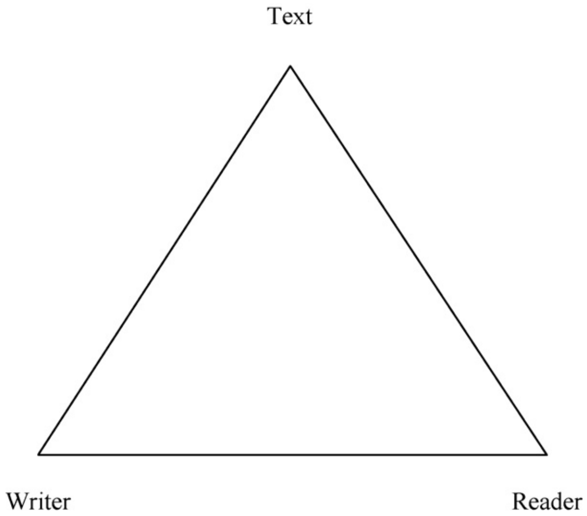

Fig. 1. Rhetorical triangle.

Based on these initial thoughts comparing the rhetoric of composition to the rhetoric of composing, we can begin to see a structure emerging. One simple place to begin mapping this new model would be to extend the comparison between composing and coding to the rhetorical triangle, assigned to various rhetoricians but usually thought to have originated with Aristotle (see Fig. 1).

But it is easy to see how the triangle would be modified by the relationship I have discussed above to look something like this (see Fig. 2). Several questions emerge from this structural comparison, one of which is the role of the editor. Traditional composition processes often involve an editor, or someone who will read and evaluate material before it is turned over

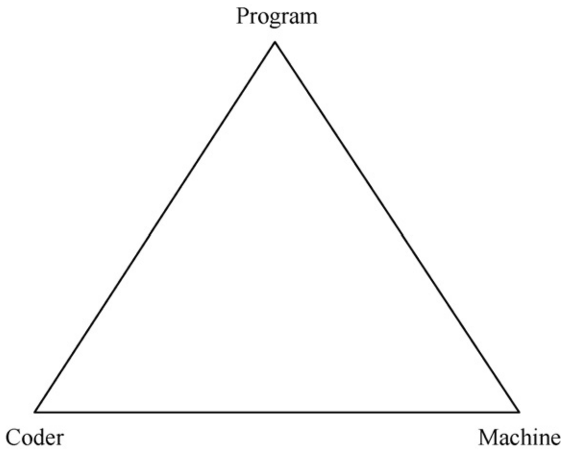

Fig. 2. Coding triangle.

to final readers. Similarly, the coder has a threshold with the machine's compiler, a software program that converts the writer's text from a high level computer language to a numerically based format that the machine can process. And in both situations, the writer's text must be processed by the editor/compiler before it can be passed on to the audience.

The role of editors also points up another problem with the comparison between writers and coders sketched thus far: Writers and coders work at differing levels of professional demand and competence. Professional writers are much more likely than student writers to have a professional editor to satisfy; the professional editor might act as a gateway to publication, holding absolute power of acceptance or rejection while a student writer's relationship with a writing instructor would merge the role of reader and commenter. But more significantly, the professional programmer might differ from the student programmer in his or her construction of an audience. While the student programmer is much more likely to view the machine as the audience for his or her writing, as it is the machine that must first react to the text, the professional coder is more likely to envision the software user as his or her audience. Student writers and coders alike are correct in constructing an audience out of teachers and machines since they will read and react—through grading or processing—to the text. Likewise, professional writers and coders must first pass through initial thresholds before their writing is published to an audience: In the traditional writing model, professional writers' texts must satisfy some sort of editor before reaching an audience, and professional programmers must first please the machine before they can please their clients. But as he or she composes, the professional writer (and editor) will be much more likely to construct and curry to a public audience, an abstraction that indeed often eludes student writers who cannot envision an audience beyond the writing instructor.

Similarly, the professional programmer who writes software that only pleases a machine will soon be looking for a new job. Exemplary computer programs work for machines and humans. Popular programming guides clearly articulate the need for programmers to produce text that is readable not only by machines but humans as well since machine language is subject to obsolescence, and there will always be a need for a human audience member to oversee the operations of the machine. This move toward a machine code that retains a parallel human readability is articulated in Andrew Hunt and David Thomas' *The Pragmatic Programmer* (2002). They have explicitly encouraged plain text programming and eschewed expressions that are needlessly obscure to human readers: "We believe that the best format for storing knowledge persistently is *plain text*. With plain text, we give ourselves the ability to manipulate knowledge, both manually and programmatically, using virtually every tool at our disposal" (emphasis in original). They continue to define plain text as "made up of printable characters in a form that can be read and understood directly by people" (p. 73).

Thus, professional coders, like professional writers, must envision an audience beyond their immediate machines and editors to produce successful prose. But our rhetorical triangle model still holds for professionals just as it does for students, for professional and student coders and writers alike must satisfy the immediate audience before reaching a distant public. The first reader for the professional writer is the editor while the professional coder must satisfy the machine as reader before reaching the client. While it remains true that both professionals have a more complex construction of audience than the student, the professional is not free of satisfying the immediate audience before reaching the larger public audience.

2. Audience invoked and audience addressed

The implications of applying lessons from coding experience—writing for the machine—to the experience of traditional composition—writing for humans—is nothing less than sweeping. Centuries of rhetoric and composition studies can be viewed in a new and instructive light if we are willing to consider that there are some instructive parallels to be explored. Even if one rejects the whole-hearted juxtaposition of the rhetorical triangle with a coding triangle, we have to acknowledge that what occurs in the coder's mind as she or he envisions the machine's reception of the program can inform much of what composition studies have had to say about how the writer envisions his or her audience. One place to begin to see the potential for this kind of outline is with one of the touchstone statements about how a writer envisions his or her audience, Walter J. Ong's "The Writer's Audience Is Always a Fiction" (1975). By interjecting an awareness of the rhetoric of coding to Ong's construction of audience, one can gain a sense of how coding can re-inscribe and complicate our construction of just one tip of the triangle.

Ong began by reminding us that the although the concept of the writer's audience may be familiar, it is relatively unexamined. One reason for this is that attempts to reveal the workings of the writer's audience have been classified as inquiries of rhetoric—and Ong found rhetoric's roots in spoken communication to be a hindrance to understanding how a writer envisions audience. Ong distinguished spoken and written communication by noting that listeners may interpret speech as "part of present actuality" and have its "meaning established by the total situation in which it comes into being" (p. 10). And since composing enjoys no such context for interpretation but can reach a potentially limitless audience—or certainly the audience that the writer is unable to foresee—Ong observed that the act of writing "normally calls for some kind of withdrawal" (p. 10). In fact writers must create an abstract conception of their audience, which Ong labeled "readership" (p. 11). And Ong reasoned that the way writers conceive of this readership is to fall back on models from earlier experiences reading fiction. Thus, Ong created a student, who is called on to write an essay entitled "How I Spent My Summer Vacation" and does not envision the audience to be his or her teacher, parent, or fellow student but rather the voice of the narrator from a work of canonical fiction. Having experienced the "authorized" success of this fiction as reader, the student writer constructs a mental audience designed to mimic the reverse experience of reading a text: "He knows what this book felt like, how the voice in it addressed its readers [. . .] why not pick up that voice and, with it, its audience? Why not make like Samuel Clemens and write for whomever Samuel Clemens was writing for?" (p. 11). His next assertion is that the text's reader is cast into a fictional role created by the author: "A reader has to play a role in which the author cast him, which seldom coincides with his role in the rest of actual life" (p. 12). Ong continued to complicate this vision of the writer's audience being based on the reading of fiction, but he never abandoned it.

On its face, Ong's construction of the reader would not seem to correspond well with the computer programmer's model of coding. One would be hard pressed to conjure examples of coders constructing audiences for their programming based on *Huck Finn* or attempting to establish an intimacy with the audience based on Hemingway's use of personal pronouns. Composing text and composing programs are certainly not identical processes. But if we consider why, in Ong's estimation, the writer goes to the trouble to construct an audience in the first place, then we can see that the underlying motivations are nearly identical. According to Ong, the writer constructs his or her fictional audience to substitute the immediate response of a physically present audience.

The coder must perform the same act, but instead of gauging human reaction, she or he anticipates the machine's reaction. There is no canonical fiction to inform the programmer during his or her coding experience unless we think of prior coding experiences as "fictions." Certainly any coder can recall the drama of watching the machine react to the programs he or she has submitted to the machine for processing, including the irritation of parsing lines for mis-keyed characters and the bewilderment of watching a processor hang indefinitely. These performances to the coder's previous texts create a library of responses within his or her memory, and the coder draws upon them just as the writer draws upon feedback received from prior texts, whether or not the coder's memories were emotionally charged (like most traditional writers, coders often rebuff the idea that the rejection of their text is an emotional experience). Additionally coders, like readers of fiction, are affected by other programs they have observed. A coder is just as likely to recall his or her human experience as user of a graceful piece or programming and allow that feeling to drive code authoring as is Ong's hypothetical student likely to recall *Huck Finn* in composing "How I Spent My Summer Vacation."

Thus, the idea of audience informs both pursuits. The coder's experience of audience within the coding triangle may not be driven by memories of being a reader of fiction, but perhaps that is as we should expect it. It is much more likely to see how the experience of using artful software, the rough equivalent of reading good writing for the writer, would drive the coder's creation process. The argument made here is not that the coding and rhetorical triangles need to be identical in order to inform each other, only parallel. Ong's assertion of fiction's influence on writing does raise another important way in which the coding and composing triangles differ yet maintain informative parallels: through the reader's conception of the author.

As Ron Fortune and James Kalmbach have recounted in their introduction to the 1999 special issue of *Computers and Composition* dedicated to the application of code in the composition classroom, the application of insight between coding and composition has been scattered and frustrating. In describing the genesis of the special issue's conception, Fortune and Kalmbach wrote that "we saw that the role of syntax and logic in writing parallels their roles in programming. Together, the two focus on organizing the internal structure of individual statements and the interaction among statements to create an effect. This is the obvious part of the connection, and although it struck us as interesting, it wasn't entirely clear what one might do with it" (p. 319). There are no easy answers. But let's start with some basic observations from the coding side and see if they can be translated into composition pedagogy.

Computers respond to writing in a way that simultaneously excites and frustrates coders. When the audience is a machine, reader response can be immediate. By hitting the "enter" button the coder immediately publishes his or her writing to a machine audience. That audience often responds in ways that are instantaneously recognized as correct or wrong. Either you get a stream of data, or you don't. The machine makes for an audience that immediately rewards or punishes the writer.

At least, that's the initial experience. In fact, once the neophyte programmer begins debugging and digs back into his or her code to find the "error" that held the program back from either compiling or processing *as intended*, he or she learns that the machine's response to his or her text was, if the term may still be applied, "correct." In fact, it could only be so: The machine responds exactly and precisely to the text as entered. There can be no other possible response, for the machine cannot think: It can only respond to the instructions that the coder has programmed for it. Therefore, it does not read the programmer's text at all. It only processes that code once the language-like syntax instructions has been compiled into numerical instructions to switch data in and out of various memory addresses.

But, in some regards, this distinction serves to reinforce the power of using the rhetorical triangle for both coding and composing. Readers of traditional text also don't respond to the ideas that the writer had when composing text. They, like the machine, must process the text on the page rather than the idea in the writer's head. The human reader reading is certainly much more complex and undefined than the machine processor, but the human and machine both cannot respond to the writer's intentions. They can only respond to the text. Thus, it is easy to understand how the computer—a machine, fast, obedient, and without intelligence—can be an invaluable tool for teaching precision in writing. And just as often as it happens that the coder is surprised in the response of the machine, so too is the writer surprised with the response of the reader.

This relationship between an author, the author's intentions, and the text-rendering device is the subject of a great deal of new media study attention. Friedrich Kittler in both *Discourse Networks* and especially in *Gramophone, Film, Typewriter* scrutinizes this relationship. The English translators of the latter work helpfully summarize Kittler's arguments as follows:

> "What distinguishes the post-Gutenberg methods of data processing from the old alphabetic storage and transmission monopoly is the fact that they no longer rely on *symbolic mediation* but instead record, in the shape of light and sound waves, visual and acoustic *effects of the real.* 'Gramophone' addresses the impact and implications of phonography, 'Film' concentrates on early cinematography, and 'Typewriter' addresses the new, technologically implemented materiality of writing that no longer lends itself to metaphysical soul building" (emphasis in original, p. xxvii).

Our construction here of the role of the computer in the creative consciousness of the programmer offers an interesting blend of these three elements. If Kittler "relates phonography, cinematography, and typing to Lacan's axiomatic registers of the real, the imaginary, and the symbolic" (p. xxvii), would the programmer's conception of his or her code not represent a commingling of these three historically contingent discursive networks? The coding process, when viewed as an act of writing, captures this "registering of the real"—without intervening authorial intent—which Kittler associated with all three technological media constructs. When the coder conceptualizes his or her desired outcome and then renders those concepts in program code, the act of programming invokes Kittler's typewriter—the reduction of imagination into barest linguistic signs.

But the subsequent rendering of that code by the machine invokes both aspects of the gramophone and film. The thoughtless and obedient machine behaves as the gramophone by faithfully rendering what it "hears;" that is to say that the computer faithfully enacts the programmer's code without regard to authorial intent, just as the gramophone produces a faithful record of audible events without regard to the human recorder's or performer's intent. But perhaps more profoundly, Kittler's analysis of film invokes the Lacanian mirror and thus introduces intentionality and recognition into the act of machine programming. For Kittler, film links the imagination of the film creator into the processing of multiple singular images, offering a viewing of the creator's imagined image similar to the image of self which confronts the infant. Just as Lacan's startled and emerging infantile self must reconcile the mirror image of herself with her mind's eye, so too must the programmer reconcile the juxtaposition of the machine's reaction to her programmed code with her desired intent. Thus, Kittler's insistence on viewing texts as material and historically situated events provides a theoretical framework for positing the desires and results of the programmer within the framework of composition.

When we consider the numerous similarities between traditional writing environments and coding environments, it is surprising that we have not experienced a call for the rhetoric of coding long before now. In many ways, this conclusion has always been present in the base assumptions of coding and rhetorical theory alike; early hints lie dormant in the work of rhetoricians as unlikely as Peter Elbow when he obliquely compares the act of writing to the mathematical process (p. 153). Perhaps it is because we tend to associate computing with mathematics that we have ignored the similarities to writing. But when we consider the rhetorical situation, as Lloyd F. Bitzer has defined it—especially as a response to a problem—then we can readily see that the application of rhetoric to the art of programming is long overdue.

3. Using code to teach writing

As composition scholars and writing teachers, we should introduce coding into our classrooms. It need not be viewed as an extensive distraction into the world of coding. Rather, a unit of coding could run parallel to composition. Indeed, this has already been done—Alfred Kern (1987) created a class at The Air Force Academy that combined the goals of teaching grammar, logic, and BASIC into one unit: "By mid-term we were using two languages—the language of grammar and the artificial language of BASIC—simultaneously if not interchangeably" (p. 5). Kern's work could serve as a model for integrating code into the classroom. His approach in one class was to use programming as a challenge to students in that they were charged with the responsibility of making their machines respond to grammatical constructions. Their first response from the computer readers was that

> [t]he students were first served with something called "syntax errors." After thirty minutes of noisy hubbub, [Kern] explained some of the misdeeds that computers call "syntax errors." Then [he] said, "Up to now, somebody has always accommodated *your* syntax errors. [. . .] But from this moment on, you have to accept the burden of understanding your computer. You are in charge, and you are going to have to correct the syntax errors." [. . .] They were the programmers; it was they who would have to teach the computer the rules of grammar. (p. 5)

Thus, Kern operates by the principle that the pre-requisite for teaching the rules of grammar to a machine is a clear understanding of those rules.

But as teachers of writing, we should not be content to use computer programming solely as a grammar tool. We can also use coding to teach a passion for revision. As has been discussed herein, one of the common statements of coders and writers alike is that they both enjoy their craft because of its ability to help them re-envision problems in a new light. While Kern's use of coding in the classroom involved the use of high-level machine instruction code, the advent of markup technology has introduced an entirely new realm of coding that is even more accessible than instruction code for the student programmer and the neophyte coding instructor. Teachers of composition need not employ programming languages specifically to focus on teaching revision; the universal truth lies in that both coding and composing enable writers to see their subjects in a new light. But the advent of markup languages like XML that allow students to begin with a base of traditional language and modify it with computer code have made the act of porting computer programming languages into the composition classroom even simpler than the dozen BASIC commands that Kern employed in 1987.

Serious, and perhaps insurmountable, obstacles hinder the application of coding to the composition classroom. Many writing instructors feel overwhelmed with the technology demands already placed upon them by the necessity of integrating the Web into writing environments. Learning a second programming language seems akin to learning French in order to make a short point or two about English grammar: The investment of research time and energy to master a programming language does not, at the time of writing, seem central enough to the mission of teaching writing to warrant the extra time. And no one needs to be reminded that technology changes rapidly: If composition teachers were to select a programming language for inclusion into the pedagogy of teaching writing, it would need to be a stable and robust language that is freely accessible to all and that also minimizes risk of obsolescence while maximizing the applicability of the programming knowledge in other classes. Therefore, it is only appropriate to conclude with some specific guidelines of how we can shape the intersection between coding and composing. A starting set of guidelines would be as follows:

- Start small

  View the inclusion of computer coding into the writing classroom as a process with an open timeline. You don't need or want to change overnight the successful composition pedagogy that you've developed during your teaching career. Focus on developing small coding exercises that parallel the writer's construction of a readerly audience and layer these in with other writing assessment tools (e.g., peer response).
- Disavow coding expertise

  A familiar cliché holds that the definition of a computer expert is someone who knows something you don't. Therefore, everyone is both an expert and a neophyte. This is still a writing class, and you're still a writing teacher. Even if you have a lot of coding experience to share, remember that the coding portion of the classroom content is to be used as a tool for better writing.
- Be clear with students about the grade consequences for the coding portion of the class

  Even though you're not required to claim coding expertise in order to include it in the composition curriculum, you will need to be clear with students as to exactly how much coding will be taught and what effect that will have on their course grade. One option is to begin with your traditional writing assignment and then layer in coding as you analyze the writing.
- Work with whichever coding language you feel most comfortable

  Since the coding portion of the classroom is a tool for better writing, you need not worry about using a current or trendy processing language. While it would be beneficial to offer the added benefit of starting students in a language with immediate applications beyond your classroom, it's not necessary. If you have more experience with LOGO or BASIC, then don't feel compelled to learn Java in the four weeks before you offer your course.

- Minimize transition time between the computer coding portion of the class and the writing portion

  The more you can integrate these two functions, the stronger the course will be. Use of a computer lab is ideal where you can minimize any transition between editing the text and using coding exercises. If institutional constraints rule out use of a computer lab during class time, consider incorporating coding into the students' homework exercises.
- Consider using markup languages

  Markup languages, such as SGML, HTML, or the ever-expanding XML, take traditional text and "mark" it with computer code. Until the development of XML, the code portion of the marked-up text was usually intended to affect how the traditional language was displayed, but XML has expanded that content to include how that language can be processed by machine. At the English Department of the University of Georgia, we found markup to be so helpful in teaching writing that we developed an XML-based application named EMMA (Desmet, 2002). EMMA is designed, among other tasks, to teach students how to markup their writing for machine processing.

Markup languages might not be coding languages in the purest sense because they are designed to add meaning to traditional text. But my class prefers XML as it allows us as a class to create our own tags. Therefore, if during a particular unit we decide to focus on the relationship between paragraph topic sentences and an essay's thesis, we can create our own XML terms to "tag" those elements in an essay. Then the XML editor parses the marked-up essay to ensure that it obeys our grammatical rules. XML offers a great deal of flexibility. Because the coding terms the student is asked to learn are the very ones the teacher introduces, or, better still, ones that the class invents, the coding language terminology is less likely to be received by the student as exterior to the writing process.

Regardless of the coding language you choose for teaching writing, however, one basic principle should guide your pedagogy: The act of writing for the machine and writing for a human audience develop similar skills, and one experience can be harnessed to inform the other.

Once, while watching an overhead projection of a computer screen in a darkened classroom, this author had the rather common experience of watching someone debug a perl script. It was during the middle of a presentation, and an unfound error had brought the presenter to a halt. We were a room full of budding programmers and so, rather than grow disgruntled and impatient, we sympathized with the presenter and made a joint project of finding that one error. A comma was out of place. Once repositioned correctly, the screen sprung to life with waterfalls of data. In silence, we all smiled as the presenter quietly said "Now that's writing with power."

Indeed, coding is writing with a newfound power. That speaker's reference to Elbow's work was not coincidental. Teachers of writing in the Digital Age should interpret the statement not as a challenge to the way Elbow and expressivist scholars have conceptualized composition but rather as a friendly supplement, as an invitation to rethink the "power" of that phrase. We can legitimately characterize that power in a cultural sense, as Herbst is wont to do, or in the technical sense, as Abelson and other programming authors would suggest. But the time has arrived for composition scholars to claim coding as their own. It is writing. Once we are comfortable with that idea, then we will find a new arena for the excitement of composition.

Robert E. Cummings received his Ph.D. from The University of Georgia and is Assistant Professor of English and Director of First-Year Composition at Columbus State University in Columbus, Georgia. He studies both American literature and Computers and Writing, and particularly their intersection through sites such as the American Literature Wiki. He has recently co-authored essays on XML for *Literary and Linguistic Computing* and *Readerly/Writerly Texts*. He is currently co-editing a volume of essays examining the advent of wikis in education entitled *The Wild, Wild Wiki: Unsettling the Frontiers of Cyberspace* (U Michigan P, 2007).

References

Abelson, Harold, Sussman, Gerald Jay, & Sussman, Julie. (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge, MA: MIT.

Baron, Dennis. (1999). From pencils to pixels: The stages of literacy technology. In Gail E. Hawisher, & Cynthia L. Selfe (Eds.), *Passions, pedagogies, and 21st-century technologies* (pp. 15–33). Logan, UT: Utah State.

Bitzer, Lloyd F. (1968). The rhetorical situation. *Philosophy & Rhetoric*, *1*(1), 1–15.

Computers and writing (2002). Teaching and learning in visual spaces. Illinois State University, May 16–19, 2002. Roundtable D.7: Can we publish essays the way open source programmers publish code? Retrieved December 13, 2002, from <http://lilt.ilstu.edu/english/cw2002/page3.html>.

Desmet, Christy (2002). Bringing up EMMA: Developing writing software with XML at the University of Georgia. Retrieved February 21, 2003, from <http://www.english.uga.edu/freshcomp/EMMA.pdf>.

Elbow, Peter. (1998). *Writing without teachers* (2nd ed.). New York: Oxford.

Fortune, Ron, & Kalmbach, James. (1999). Letter from the guest editors. *Computers and Composition*, *16*(3), 319–324.

Herbst, Claudia (2002). Blood, sweat and code: A new text, power and illiteracy in the context of gender. *The Journal of Literacy and Technology*, *2* (1). Retrieved November 21, 2002, from <http://www.literacyandtechnology.org/v2n1/herbst.html>.

Hunt, Andrew, & Thomas, David. (2002). *The pragmatic programmer: From journeyman to master*. Reading, MA: Addison-Wesley.

Kern, Alfred. (1987). Basic writing: the student as programmer. *ADE Bulletin*, *86*, 4–7.

Kittler, Friedrich (1999). Gramophone, film, typewriter (G. Winthrop-Young and M. Wutz, Trans.). Stanford: Stanford University Press. (Original work published 1986).

Knuth, Donald E. (1997). *Preface. The art of computer programming* (3rd ed.). Reading, MA: Addison-Wesley.

LeBlanc, Paul. (1993). *Writing teachers writing software: Creating our place in the electronic age*. Urbana, IL: National Council of Teachers of English.

Modern Language Association. Call for action on problems in scholarly book publishing. Retrieved October 7, 2002. from <http://www.mla.org/www_mla_org/REPORTS/pdf/presletter6_02.pdf>.

Ong, Walter J. (1975). The writer's audience is always a fiction. *PMLA*, *90*(1), 9–21.

Ramsay, Steven (2002). Algorithmic criticism. Unpublished Doctoral Dissertation, University of Virginia, Charlottesville. Retrieved February 21, 2003, from <http://cantor.english.uga.edu:8080/algocrit/servlet/ramanujan.AlgoCrit?xml=algorithmic.xml&xsl=algorithimic.xsl>.

Stallman, Richard (2002). GNU Project, Free Software Foundation. What is copyleft? Retrieved October 5, 2002, from <http://www.gnu.org/licenses/licenses.html#WhatIsCopyleft>.