MVC, Materiality, and the Magus: The Rhetoric of Source-Level Production

"The greater the range and intenseness of the opportunities for the exercising of our symbolic prowess, the greater might be our delight in such modes of action." Kenneth Burke, "Rhetoric and Poetics," *Language as Symbolic Action* (1966, 297)

Let me open with the major premise of this chapter's argument: programming is writing. I mean that literally. Yes, "programming is writing" is not infrequently invoked as a metaphor. And I agree with programmer Steve McConnell (2004) who dismisses writing as "the most primitive metaphor for software development," although I don't agree with his reasons for dismissal. Among McConnell's observations on the metaphor's shortcomings:

- "writing doesn't require any formal planning, and you figure out what you want to say as you go"
- "writing is usually a one-person activity"
- "in writing, a high premium is placed on originality. In software construction, originality is often less effective than focusing on the reuse of design ideas, code, and test cases from previous artifacts"
- "when you finish writing a letter, you stuff it into an envelope and mail it. You can't change it anymore"

Rhetoricians will immediately recognize the oversimplified conception of writing at the heart of McConnell's critique of the metaphor. True, writing may not require formal planning, but as even the introductory-writing student quickly discovers, that's an unstudied and ineffective way to proceed. We know that writing is collaborative, to some degree, always. And from citation practices to genre features, writers have a wide foundation of reusable material upon which "originality" is built with no small degree of effectiveness, given a particular rhetorical occasion. To write is to engage in intelligent, ethical (i.e., non-plagiarized) reuse of "previous artifacts." (However, as I will discuss later, in digital writing we reuse and outsource production with reckless abandon, preventing richer encounters with "originality.")

Granted, in certain forms of writing on certain occasions, change becomes impossible or at least very difficult, in the way McConnell suggests. But even digital writing of any complexity beyond a word processor document typically fares no better when it comes to revision and

realizing the possibility of ongoing change. One-off digital invention–as in the first (and only) stab at writing even a basic web page–will have to suffice for the digital rhetorical act. There is simply too much risk involved in revision when digital production occurs apart from source code.

Materiality

Just as McConnell oversimplified the activity of writing, so too do writers fall prey to oversimplifying the activity of programming. When I claim that "programming is writing," I am talking about writing source code right alongside or in service to natural languages. More than 30 years of visual interfaces interceding on behalf of writers and language (computer and natural) have arguably almost fully obscured the rich symbolic activity happening just beneath even the apparent simplicity of an iPad's touch screen. The remaining symbolic activity that hasn't been obscured is perceived to be simply beyond our symbolic concerns as writers and rhetoricians. What could be more preposterous than writers programming, when an app store app or an open-source package is readily available for download and installation? Shouldn't writers, as Bill Hart-Davidson (2012) has suggested, simply outsource programming to other people?

Such questions belie an impoverished, even ham-fisted conception of what programming is and what it does, or can do. Those questions arrest any consideration of the symbolic materiality and activity at programming's core. They discourage the deep, long-term involvement with the digital medium that I believe is necessary to engage programmers and developers who are doing so much of our writing, so much of our rhetoric–even when we think, as Hart-Davidson seems to, that programmers can simply translate/program *our* ideas and make them reality.

Jay David Bolter's simple 1991 observation seems all but forgotten today: "Even a graphics program does not draw: it writes." Working at a time even before the introduction of Microsoft Windows 3.1 and the mouse-and-GUI model of interaction that that operating system would make ubiquitous, Bolter could confidently proclaim what seems tenuous today: "All computing is reading and writing. The computer is therefore a technology for all writers–scientists and engineers as well as scholars, novelists, and poets" (10).

It is no huge leap to reformulate Bolter's observation as "All *programming* is reading and writing." Of course, outside of small groups of individuals in the field of rhetoric, particularly within Computers and Writing, the idea that "programming is writing," or even that programming counts as intellectual work, is far from accepted.

So it is within the broader pursuit of digital humanities. Ramsay and Rockwell (2012) note that, for those "who have turned to building, hacking, and coding as part of their normal research activity," there is a looming question of "whether the manipulation of features, objects, and states of interest using the language of coding or programming…constitutes theorizing" within the digital humanities. Although far from arriving at any actionable answer to that question, Ramsay and Rockwell articulate the challenge that faces anyone who would claim that programming is literally writing: only those who program and build are able "to present their own activities as capable of providing affordances as rich and provocative as that of writing."

The key word there is "activities." It is not simply the created artifact, or the source code behind it, providing the affordances of intellectual work. It is the activity of programming itself that builders, writers-as-programmers must present. This activity, I believe, will ultimately present itself as "theorizing" as Ramsay and Rockwell seem to hope. But the path to theorization may very well look different from simply demonstrating to others a "rich and provocative" set of affordances meant to *metaphorically* suggest "programming is writing."

Rhetoric's roots as a practical art, embracing theory as well as techne and craft, must be brought to the fore in order to render the practice of programming as a knowledge-generating, epistemological activity within the province of rhetoric (and the broader digital humanities).

Malcolm McCullough observes,

> As we overcome the residual notion that computing is for objective documentation only, we must cultivate expressive sensibilities. These may result in a digital aesthetic or poetics…. And in the end, chances are that appropriate artifacts and descriptions will engage us through rich and transparent tools, built on newfound densities of symbolic notation and personally experienced as a medium. (1996, 219)

A fully realized, nuanced digital rhetoric has been arrested by literary modes of knowledge-making (as represented by Ramsay and Rockwell's fixation on "theorization") on the one hand, and the WYSIWYG interface on the other. In fact, I don't think a rhetoric of programming is possible to articulate at the moment; there are simply not enough of us who program. At the same time, the "personal experience" of a medium that McCullough calls for is lost by relentless outsourcing of what could otherwise be rhetorical knowledge-making work; writers outsource that activity to programmers, to WYSIWYGs, and to readymade

software packages like WordPress and Drupal.

For those of us who do program as an inherent part of our research agendas, our argument must proceed by demonstrating that programming is genuine inquiry that resists denigration with regard to any other knowledge-making activity rooted in the manipulation (and interpretation) of symbols.

From my point of view, this draws us in close company to the knowledge-making practices of art and design. In an obscure but important pamphlet published by the Royal College of Art, Christopher Frayling (1993/4) urges differentiating between three craft-oriented modes of research: research *into* art and design; research *for* art and design; and research *through* art and design.

Rhetoric, composition, and technical communication has no small body of work that researches *into* writing, often by studying writers and their contexts: for example, introductory writing students in first-year composition to seasoned technical writers working in the public or private sector. The research conducted *for* writing often takes a pedagogical turn: we research into making writing more teachable for others (students) rather than to necessarily improve our own practice.

But it is Frayling's last prepositional category, research *through* writing and rhetoric–indeed through programming, as "personally experienced" symbolic activity–that remains the great undiscovered and currently undervalued territory of knowledge-making. Programming remains a mysterious and magical activity.

Magic & Programming

Programming is writing. Programming is symbolic activity, in other words, within the province of rhetoric. To program, as to write, is to make and act in the realm of the symbolic: "making language has long been regarded as, in some sense, magical; as a *spell*" writes William Covino in *Magic, Rhetoric, and Literacy*, adding "spelling, even in its ostensibly nonmagical sense, denotes the visible materialization of invisible thought" (1994, 5).

Decades before Steve Jobs pronounced the iPad to be a magical device at its unveiling in 2010, Arthur C. Clarke had already articulated what has become known as Clarke's Third Law: "Any sufficiently advanced technology is indistinguishable from magic." It is no feat of imagination to see the magic inherent in multitouch interfaces, retina displays, ubiquitous high-speed internet connectivity, and other hardware-based wonders introduced over the last

decade.

But the magical incantation of the written word in one programming language or another is what should make the symbolic activity of programming a particularly interesting problem for rhetoric.

Recalling the scene from *The Matrix* where Neo announces to Tank from inside the Construct that he needs “Guns. Lots of guns,” digital designer John Maeda describes “the sense of magic that occurs when Neo expresses his wish. The instantaneous rush of tremendous resources, as visualized in the simple special effect of this scene, epitomizes for me the experience of freedom when programming the invisible spaces of computer codes” (2004, 17).

Where Maeda personally experiences “freedom,” of course, other people no doubt experience the fear inherent in contemplating the magical symbolic activity of programming. Fear that stems from closing ourselves off from personal experiences of the digital medium realized through programming. “Fear of magic has always been with us, in particular the fear of magic words…which claim to define or alter reality” (Covino 1994, 1).

Yet “Even in our nonmagical world, magic remains a conceptual construct for appraising the powers of language, and while magic is understood by some as a liberatory alternative to established rationalism, the prospect of a magical epistemology is a fearful one even for those who endorse literacy as a disruptive force” (Covino 1994, 5). Programming presents itself as magic to the uninitiated. I have witnessed that even the simple prospect of programming can arrest into silent hostility rhetoricians and writers who otherwise “endorse literacy as a disruptive force” and who ostensibly delight in an ever greater “range and intenseness” of symbolic activity (as Burke 1966 suggests in the epigraph to this chapter).

But magic is not simply magic in Covino’s treatment. In what Covino describes as “arresting magic,” programming would indeed by conjured only by a particular magus, the programmer, whose spells and incantations (realized in technologies from word processors to smart phones) represent “the imposition of the powerful few upon the unquestioning many” (1994, 8). However, magic presents itself in another sense: “the practice of disrupting and critiquing articulate power: a (re)sorcery of spells for generating multiple perspectives.” Noting that “generative magic enters the world it questions,” however, Covino argues that generative magic is “an amplification of the possibilities for action” (8).

How are we to discover those possibilities for action, if not through extended personal

encounters with programming ourselves? We can conduct empirical studies of writers and programmers, we can try out different methods of teaching these techniques to students–but unless and until our body of research inside of rhetoric and alongside the digital humanities allows for research *through* programming, we will be no closer to creating the knowledge that will push our field into broader conversations and even be leaders with regard to the symbolic action made possible by programming.

Visual interfaces, particularly those for digital production and expression, are readymade materializations of invisible thoughts of programmers. By denying that programming is writing–either outright or by the outsourcing of programming to WordPress or Drupal, or to some hired programmer–writers are compelled to compose through other people's interfaces, other people's text boxes. Programming is someone else's problem in this scheme, and our understanding of it is limited and impoverished.

Model-View-Controller

To speak of "programming" in a vague, monolithic abstract is no different from speaking of "cooking" or even "writing." Any encounter with those activities is always already more complex than the monolithic category itself.

Although I applaud the intellectual curiosity underlying books such as *10 PRINT* (Montfort et al. 2013) and certain other pieces of scholarship emerging from software studies, too often I have encountered conference presentations and scholarship–including *10 PRINT*'s focus on a single line of BASIC–that showcase truly ancient source code that neither looks nor functions in any way that's recognizable from contemporary programming languages such as JavaScript, Python, or Ruby.

Not only have the syntaxes of programming languages evolved, but so too have the paradigms and idiomatic forms within given languages. For the remainder of this chapter (which continues on in digital form at GitHub), I want to provide a brief overview of a specific software construction paradigm, model-view-controller (MVC), and the almost religious devotion to MVC inherent in the Ruby on Rails web development framework.

In MVC, there is a recognition of the three essential components of any digital system. The model is concerned with data in the abstract; ensuring, for example, the correct number of digits in a credit-card number. The view is concerned only with the presentation, typically visual, of the data in the model: outputting, for example, an editable field with a credit-card number–or perhaps an obscured version of the number: `**** **** **** 4567`. And

finally, the controller awaits commands from a user seeking to, in this example, enter or revise a credit-card number, and perhaps ultimately make a purchase using the credit card. The controller may send data to the model, or make requests from the model for certain data that will ultimately be rendered in the view.

I won't go into a lengthy technical description of Ruby on Rails (just search the Web for it), but I will say that it is a Web development framework written in Ruby. Rails is installed on a computer via Ruby gems, a software-package manager for the Ruby programming language, which Rails is written in.

Rails should be of interest to writer-programmers for many reasons, but my primary purpose in showcasing it here is that Rails can be installed, invoked, and developed using only writing. There is no file to download and unzip (like Drupal, WordPress, and other platforms-as-frameworks that are currently popular). Rails is installed by running a command on the command line (indicated throughout this chapter by use of the dollar-sign, `$`):

```
$ gem install rails
```

But installing Rails does not directly create a Rails application. Rather, having Rails installed provides access to its library of different modules, including those for handling models, views, and controllers. Rails also includes a command-line program, conveniently called `rails`; and it is with the `rails` command that a project is brought to life from the text-based command-line interface (CLI).

The conclusion of this chapter will be found in the Rails app hosted on GitHub at https://github.com/xxx/xxx.git

Bibliography

Bolter, Jay David. *Writing Space: The Computer, Hypertext, and the History of Writing*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1991.

Burke, Kenneth. *Language as Symbolic Action: Essays on Life, Literature, and Method*. Berkeley, CA: University of California Press, 1966.

Covino, William A. *Magic, Rhetoric, and Literacy: An Eccentric History of the Composing Imagination*. Albany, NY: Statue University of New York Press, 1994.

Frayling, Christopher. "Reseach in Art and Design." *Royal College of Art Research Papers* 1,

no. 1 (1993/4).

Hart-Davidson, Bill. “Code? Not so Much.” *Gayle Morris Sweetland Digital Rhetoric Collaborative* (blog). http://www.digitalrhetoriccollaborative.org/2012/10/17/code-not-so-much/

Maeda, John. *Creative Code*. New York, NY: Thames and Hudson, 2004.

McConnell, Steve. *Code Complete*. 2nd ed. Redmond, WA: Microsoft Press, 2004. Kindle edition.

McCullough, Malcolm. *Abstracting Craft: The Practiced Digital Hand*. Cambridge, MA: The MIT Press, 1996.

Montfort, Nick, Patsy Baudoin, John Bell, Ian Bogost Jeremy Douglass, Mark C. Marino, Michael Mateas Casey Reas, Mark Sample, and Noah Vawter. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. Cambridge, MA: The MIT Press, 2013.

Ramsay, Stephen and Geoffrey Rockwell, “Developing Things: Notes toward an Epistemology of Building in the Digital Humanities,” in *Debates in the Digital Humanities*, edited by Matthew K. Gold. Minneapolis, MN: University of Minnesota Press, 2012. Kindle edition.