

## Code and Composition

Luke Fernandez.  
Weber State University

Forty years ago C.P. Snow challenged academe to redress what he saw as the corrosive division of the university into two cultures – the culture of so-called "literary intellectuals" and the culture of scientists.

If this division really exists today its an amorphous one. Many scientists appreciate the interpretive approach of disciplines which hold out that much of knowledge is socially constructed. And many people in the humanities and the social sciences understand the virtues inherent in a scientific method. Humanists may tolerate a little more "fuzzyness" and will uphold the value of "interpretation" but they can certainly appreciate the value in an approach that insists on confining assertions to what can be tested and potentially proven wrong.

But if there are plenty of academics who have their feet in both cultures, it's no accident that many people still know of C.P. Snow.

That's because the vestiges of the two cultures still exist. After all, most universities still have a college of the humanities and a college of sciences, and this bureaucratic organization allows a division to fester in ways that perhaps it wouldn't were some other organization of knowledge allowed to take hold.

In subsequent work C.P. Snow hoped for the emergence of a 'third culture' that would help to bridge the divide. These 'third cultures' are evident throughout academe and they're perhaps most in evidence in interdisciplinary programs where science is explored from the perspective of the humanities and the humanities are explored through the lens of science. But the 'third culture' is also found in a less likely place: the growing cadre of university programmers (also known as coders or developers) who are squirreled away in IT division cubicles or in computer science labs. These people also inhabit the 'third culture.' Coding, like engineering, in many ways occupies a middle ground between people of letters (who use composition as a way of creating new worlds or re-interpreting existing ones) and scientists who also write essays but usually do so with the intent of precisely describing the existing world with assertions that are testable and clearly verifiable or falsifiable. Because programmers engage in science and interpretation, because they have a foot in both cultures, they can play a role in ameliorating C.P. Snow's divide.

In the following essay I describe some of the major activities that programmers engage in – or at least those that would be interesting to humanists who write or engage in composition. In so doing, I try to show how these cultures are not quite as disparate as the traditional course catalog would suggest. Courses in composition and courses in computer science currently pass each other in the catalog as ships in the night. But a large part of computer science has to do with coding; and the culture of coding and the culture of composition share enough that their commonalities deserve enumeration – at least if we're interested in acknowledging and fostering academe's third culture.

## **CODING AND COMPOSITION – Some elementary similarities and differences.**

### **Organic Languages versus Invented Languages**

Computer programmers code in a language just as writers scribe in a language. But in spite of this shared connection -- the act of recording thoughts in a medium that can be revisited long after it has been forgotten -- one should be careful not to gloss over the differences.

Unless one is writing in Latin, when one writes one is scribing in a language that is also spoken. People generally think and dream and speak a language before they begin to write it. Not so with code.

Unless I'm Spock, or the Terminator or some other form of cyborg, when I code I'm not writing in a language that I also speak in my day-to-day life. Code hangs by itself, alone, isolated, unconnected from the language I use for day-to-day living. I can talk casually with my spouse about nostalgia or envy or love, and then, in a more quiescent moment write (in English) about these things as well. But the same isn't exactly true about code. I may "talk" code with my coworkers but I don't use the language outside work. And even when I'm at work I don't read or speak long passages of code in the way that an orator may read aloud at length from a written text. Programmers rarely pick up and start reading code in as casual a manner as the way someone picks up a paper and reads it on a bus. But when a programmer does have to read somebody else's code he would rather read it than "listen" to a fellow programmer speak it to him.

I may catch myself thinking about code while I'm at dinner but I can't translate these thoughts back out into a vernacular that I share with very many others. Code isn't like this -- it doesn't provide direct access to human consciousness in the way that writing does to one's

mother tongue. [p. ong 163] And like Latin, there are 'no purely oral users' of a computer language [ p.113] Unlike composition, where literacy is tied however unevenly with orality, code has no connection to an oral culture. We can hear HAL (in 2001: A Space Odyssey) speak in a robotic voice and we make a connection between these utterances and some imagined code. But these connections are tenuous. Code is a written language but even this far into the information age it has yet to become a language that is very often spoken. Invented ex-nihilo (before it was ever spoken) it doesn't have much reason to be voiced until, perhaps, we become cyborgs.

## **Vocabulary and Emotion**

Programmers and writers are both recording thought in a written language. But whereas computer languages are invented ex-nihilo, the invention of writing exists in a distant past. People may attempt to create lexicons and dictionaries that standardize a written language but the evolution of the culture in which the language is spoken (unless it's a dead language like Latin) makes its growth more organic. Does this difference have any consequences for how coding and writing are experienced? It does. For one thing, computer languages typically have a vocabulary of only a few hundred words -- although the inventor may add more words to the vocabulary, it isn't a lexicon that is organically tied to a larger non-computer culture. So as a programmer I'm to some extent at the mercy of the people who invented the computer language I'm using -- I can't expect the larger culture in which I live to catalyze the development of new words.

That said, a good computer language allows a programmer to expand the vocabulary on his own. For example, if there isn't a word for automobile she can create this word. And if there isn't an action to describe what an automobile does, or if there isn't an adjective to describe the color of the car, she can invent words for these things and actions as well. While the originators of the computer language rarely add these words to the basic lexicon, the programmer can share these words with other programmers. A good programmer creates a vocabulary that is flexible but that also precisely describes the world she is trying to create. If she thinks her virtual world needs to have trucks and buses and cars she will first create a word for automobiles. Once she has the automobile word, she will create derivatives of that word as a way of efficiently describing other moving vehicles that share an automobile's attributes but contain a few more attributes as well. In writing, we avail ourselves of an existing lexicon, we don't generally try to expand on it (unless perhaps you are George Bush).

But when writing code programmers aren't just trying to figure out how to use the existing lexicon, they also need to know how to expand it to make it accurately describe their virtual world.

The most commonly used computer languages don't have as part of their basic lexicons words that describe human emotion. As a result, programmers, at least when working with a standard language, aren't in a position to quickly expand their emotional consciousness in the same way as, for example, learning and employing a word like agoraphobia might expand the emotional richness of a writer's life.

But even though computer languages are designed to talk to machines that aren't sentient enough to need to know a word like agoraphobia, computer languages are ultimately there so that programmers can speak to other humans through the machine. If a programmer needs to create a word like agoraphobia there isn't anything in the computer language which will prevent her from adding it (although the word would be essentially meaningless unless there are other words and a virtual contexts in which to situate it).

### **CODING AND COMPOSITION – Systematizing the experience**

If the language of code isn't as emotive as a spoken language, this doesn't mean that coding and composing are alien experiences – coders and composers are attempting to make order out of disorder and their approaches for doing this are not so dissimilar.

To write effectively I need to immerse myself in the experience – while I may know more or less what I want to communicate before I write it down, often the process of writing will make me think of things, will reveal problems or nuances that I wouldn't have become aware of until I actually engaged in the process of writing. At least in principle good coding isn't supposed to be conducted this way. Although programmers are tempted to start coding as soon as they identify a need that they would like to cater to, software is first supposed to go through a more abstracted process where the problem (or problems) that the code is supposed to resolve are identified and written down on paper, where the particular features that are needed are enumerated, and where the user interface is sketched out in drawings that would be recognizable to an end user. Only after this process has been completed are the programmers supposed to actually sit down and start coding. The idea and intents are supposed to be articulated a priori – the coding is only supposed to carry out the ends that were identified in the previous steps.

On the surface this more systematized process would suggest that writing and coding are very different activities. But the activities are more alike on closer analysis. First, writers often engage in a process of research and planning before actually committing anything to paper. A lot of writers try to outline their arguments before getting down into the thick of things. Dissertation writers have to write a proposal in which a dissertation question is identified, a methodology for answering the question is articulated, and chapters are titled and enumerated. Second, while good programmers will engage in research and design before actually coding, the actual coding often reveals problems and nuances that were not anticipated in the initial research and design. Just as a dissertation proposal is modified and revised in light of discoveries that are brought to light through the process of writing, the initial design of software is tweaked, revised and updated in the actual coding. Often a coder will code things once only to realize that she didn't really know what she was doing the first time she wrote it. Like good writing, coding generally improves through a process of drafting and redrafting. And what one really intends to make often isn't completely clear until one has written or coded a few drafts. Like a writer, a coder is intent on creating some degree of order in the world around her. Order congeals only slowly and its best character isn't always revealed until one or two coding attempts are tried and refined.

If there's a real difference between coding and writing (with respect to hanging back and thinking before actual coding or writing) it's more one of instinct than actual practice. Most writers tend to hesitate and think a long time before actually getting down to the task of writing. This may be because while writing can be fun once the writer is actually immersed in the experience, the actual process of immersion can be painful. Coders, on the other hand, tend to charge into the coding pre-maturely maybe because it's a little easier to begin the initial immersion. Where writers often need to think of the homily, "it's never too soon to start writing" coders often need to remember to hang back.

### **CODING AND COMPOSITION – Syntax, grammar, and the basic tools of the trade.**

Although some writers compose directly on a computer screen, many writers compose with pen and paper before digitizing things. The use of different technologies allows the activity of writing to take place in more than one environment and these multiple contexts, for those who use them, often improve the prose. I immerse myself and focus

best when I'm reclining in an easy chair with pen and paper in front of me. But eventually, at least after a first draft, I'm drawn to the computer because of the tools it offers me including a dictionary, a thesaurus, spelling and grammar-checking.

### *Proofreading and IDEs*

Similar, albeit more powerful tools are available to the programmer when she composes which is why the majority of coding takes place in front of the screen. Although the basic lexicon of most programming languages is much smaller than English it's sometimes difficult to remember syntactical rules. To help in this endeavor, programmers use Integrated Development Environments (among programmers these are better known as "IDE"s). IDEs are like revved up word processors. IDEs vary the color of the code font to help programmers read their code, IDEs provide ready access to the coding lexicon – when a programmer wants to use a seldom used word, the IDEs lexicon provides invaluable information about the word's etymology and the way it can be used in a sentence of code. IDEs also provide inline editing. Much like an advanced word processor's spell checker, an IDE will provide a programmer with running suggestions about the spelling and syntax of her code as she's actually typing.

When coding, a programmer talks to a machine – the grammatical and syntactical demands of the machine get internalized by the programmer the more code she writes. Just like a writer, coders are sometimes hired on the basis of how much they've scribed – the more lines of code one has scribed the more likely that the programmer has had incentives and opportunities for honing their technique. But the internalization of a computer language doesn't happen right away, and like the use of a native language there are always personal lacunae – areas in the language where one needs to be reminded of the mechanics, where one's technique is wanting. Programmers use IDEs partly because of these lacunae. An IDE is not exactly the same thing as a word processor. Coders and writers don't resort to these tools for exactly the same set of reasons. All that said, many of the reasons are the same.

### *Advanced Editing*

If coding, like writing, suffers from mechanical mistakes code is also subject to deeper, more intractable, forms of incoherence. The more elementary incoherence can be caught by an IDE. But deeper less obvious inconsistencies or incoherent aspects of the code often aren't

transparent.

While a programmer wants her words to be flexible, so that other words can be derived from these words, she also wants her words to be defined with some precision. If for example she's defined an automobile with attributes of a truck, her code may create incoherencies if she tries to derive a bus from that word. If she's lucky, the incoherencies will manifest themselves sooner rather than later. But sometimes the problems won't become apparent except in weird use cases long after the code has been distributed to customers.

For example, a customer may try to ride a virtual bus but finds out that the bus seems to have qualities that are more reminiscent of a truck. In a written language we are used to words with multivalent meanings and we can employ them without compromising clarity because of the context in which they are written. While this is sometimes true in computer languages, amorphously defined words aren't usually a good idea. They lead to problems later down the line.

To preempt these problems, coding, like writing, goes through a process of review (which in a programmer's world is called 'quality assurance') that to some extent can be likened to the process of proofreading and editing a manuscript. The code is tested and retested by other people because a second or third set of eyes often quickly uncovers problems that are invisible to the initial author.

Seldom if ever is good code produced on a first try. Quality assurance teams, like editors, examine the larger message or intents that the code is trying to convey (or the world that the code is attempting to frame) to make sure it actually does what it's supposed to do and that incoherence is minimized.

## **Collaboration Tools**

Programming is often perceived by people outside the culture as a solitary activity that attracts people with asocial dispositions. There is some truth to this. A fellow programmer once confessed to me that he was attracted to programming because he couldn't predict or understand people. He took up programming because computers seemed much more fathomable. Programming appears solitary from the outside looking in – because of the asocial repute of programmers it sometimes appears even more solitary than writing. But on the whole it is a much more social activity.

Unlike most books or essays, software isn't usually produced by a single author. It is written by teams of programmers who have to

negotiate dual worlds. On the one hand programming demands a lot of abstract and complex thought – the kind of thought that is done with a certain modicum of privacy. To gain focus, and to competently create order out of disorder, programmers often don't suffer interruptions and the social interactions that accompany them very well. Many programmers show up late at work and leave late because their most productive work happens after 5:00 pm, when the phones stop ringing and the office becomes quieter. Programmers often move into management or other types of vocational activities when they start raising families because the responsibilities of parenthood often interfere with a programmer's need for focus.

But if certain imperatives of coding impel programmers to seek solitude, programmers, unlike the majority of essayists or book authors, produce software most often in groups. Despite their asocial reputation, programmers have to work well with others in order to succeed. They have to be able to negotiate between a world of quiescence, solitude and abstract thinking, and a world of back-and-forth interaction with other people.

To negotiate these transitions effectively programmers are sometimes asked to work with project managers and system architects. These people spend time creating work environments and software frameworks that allow programmers to code at times in private while contributing that private work to a larger team effort. The challenge of accommodating and coordinating these individual and communal imperatives are not easy to master and for a long time the best way to do this was to assemble teams of programmers in the same geographic location (like Microsoft's fabled corporate campus in Redmond, Washington) so that at least the factors of a shared place and a shared time zone would encourage more communal activity.

But these tools for pursuing communitarian ends are beginning to be displaced by other technologies that can foster collaboration without putting constraints on time and place. Communitarian ends can be achieved though means that, on the surface anyway, appear as highly individualistic as the activity of writing and composition. For example, programmers will often appear to be working and authoring alone when they really are not. Programmers spend a significant amount of time in front of their I.D.E.s. But glance at a programmer's computer screens and as likely as not she'll also be running an email and an instant messaging client – she might even be on the phone. An I.D.E. is invaluable in working with a programming language's lexicon, but sometimes messaging with another programmer is a more expeditious

way of uncovering and understanding some lexical complexity. And often it's better than a phone conversation or a face to face conversation because the subtleties of the language are best communicated through writing.

Because software is often co-authored by teams of programmers, it's also critical to have an application that logs a programmer's code submissions and that also ensures that her submissions are revising the same version of the software as her colleagues. When a writer works with an editor the same challenges arise – writers and editors have to make sure they are editing the same version of an essay – but imagine how much more daunting the challenge is when working with a draft that three, ten or one hundred programmers are attempting to revise all at the same time. To accomplish this, programmers use something called "version-control software" which keeps a log of all changes and ensures that one developer doesn't inadvertently overwrite somebody else's code. On really big projects, organizations will hire "build engineers" whose full time job is to track and control software changes.

Although writing is presented with the same co-authoring problems it's not challenged to the same degree and as a result the experience is not commensurate. Still, authors can get a feel for how their more solitary composition differs from the more communal composing that programmers do. Venture out to a Wikipedia entry and click on the history and discussion tabs. These tabs track and manage the evolution of co-authored composition. Click on a former entry on the history tab and the Wikipedia user can revert the current entry to a former draft. Click on a discussion item and the Wikipedia user can uncover some of the rationales on which a particular edit or set of edits was predicated. Programmers use similar (albeit more sophisticated technologies) to manage their own collaborative and evolving compositions.

### **CODING AND COMPOSITION – Imagining and conversing with one's audience.**

Despite the need to focus on grammar and syntax it's important to note that programmers aren't only "bit twiddlers" who just need to understand a basic syntax to produce something of merit. To talk to the machine (and by extension to talk to one's end users), syntactical rules need to be followed. But correct syntax is only the lowest threshold by which code is judged. However good code may be on a syntactical level it won't have much merit if in the aggregate it isn't

creating an application that is appreciated by its eventual users. Like good writing, good code can be judged by its concision and efficiency, and on whether it abides by a larger structure that is identifiable to other programmers – convolution is a definite no-no and programmers, like writers, spend a good deal of time trying to avoid falling prey to it. But all of this is secondary to writing and coding's main intents. Eloquence is important but if a programmer's audience doesn't find the aggregate message compelling, the eloquence is hollow. For programmers to be successful they, like writers, need to imagine their audience.

If coding was just like writing, it would be a very agonizing activity because imagining one's audience is not an easy thing. As Walter Ong has observed in *Orality and Literacy* unlike speaking or talking to an audience, 'the writer's audience is always a fiction...The writer must set up a role in which absent and often unknown readers can cast themselves.' [Walter Ong *Orality and Literacy* (New York: 1982) p.102] This, according to Ong, is one of the more painful activities of writing. Coders don't suffer this quite as much. While a coder must also imagine his audience, the computer offers a companionable surrogate. Late at night, when everyone else is sleeping the computer is still a willing listener and a tireless critic. Because the programmer has a companion perhaps this is why it is easier to immerse oneself in code than in writing. Because of its sociability, (because of its essential orality), it isn't as agonizing as composition.

In our imagination we tend to envision programmers squirreled away in cubicles, twiddling away in confines far removed from social settings. The fact that programmers, when they code, are conversing with a computer rather than a fellow human would suggest that what they're doing is asocial. But in the spectrum of social to asocial activities one shouldn't presume too quickly that programming "feels" like an asocial activity to a programmer. In fact, when compared with composing an essay, the activity is in some ways more social. Writing is a monologic activity – a writer's audience isn't in the writer's immediate presence and won't respond or interrupt him in the same way that someone will when people talk face to face. Writers have to imagine their audience when they write. Coding, in contrast, doesn't feel this hermetic/solitary because it's more like talking. A programmer talks a little bit to her computer (in code) and then sees how the computer responds. If the programmer has coded clearly, and has anticipated how the computer will behave, she will get the feedback she expects. If she hasn't coded well, she'll be reprimanded with appropriate warnings "the page you are accessing has

experienced an unexpected error....the error occurred on line 65." In Web programming, which nowadays comprises a good portion of all the programming that is going on, the activity is even more social, because the coder isn't talking to just one computer but to multiple Web browsers. Although programmers lament the fact that these browsers don't all understand code the same way, this ultimately adds to the sociability of programming. When we talk to adults, we often talk one way and when we talk to a child we often talk another way. A corollary exists when Web programming – each browser demands a different intonation or 'flavor' of code – they won't do what the programmer wants unless she speaks to them differently.

Coding feels like a more social experience than most composition because programmers have a surrogate audience. If it 'feels' more social often it's substantively more social as well. Coders often author things with other coders and, if they do their homework, they'll have gathered a list of their audience's desires through "needs analysis" interviews before they begin to code anything. So coders don't have to imagine their audience quite as much as writers engaged in composition. Still, the computer is only a surrogate audience – it can only speak for the end user imperfectly. And end users usually don't fully articulate their needs – they can tell programmers in general what they want but the programmer has to fill in the details. For example, end users can tell a programmer that they want their new word processor to be able to make footnotes and this seems like a clearly stated specification. But when the programmer actually sits down to write the application she is left to fill in the gaps. "What should the program do if the footnote is really long? Should it flow to the next page? If so, after how many words should it begin to flow to the next page?" Usually questions like these are left unanswered and the programmer is left to guess or imagine how her audience would answer them. [Scott Rosenberg interviews Joel Spolsky in Salon "The Shlemiel way of software", <http://dir.salon.com/story/tech/feature/2004/12/09/spolsky/index.html>]

Despite the fact that the experience of coding often resonates with the experience of writing, one should be careful not to conflate them.

Talking to a machine is a very different kind of dialogue than the type of internal dialogue one generally has while writing. Rules and constraints are internalized by both the coder and the writer, but while a writer has to imagine his audience, a coder isn't doing this all the time. When the programmer speaks to the computer her audience is right there in front of her, giving her feedback on what works or

doesn't work on a syntactical level. Moreover, the general needs of her audience have often been discovered and specified beforehand – she doesn't have to imagine the general disposition of her end audience.

But like the writer, the quirks and idiosyncrasies of a programmer's end audience are still left up to the imagination – the programmer has to guess them. Until the finished product is placed in front of the user it's not possible to anticipate exactly how that person will react.

## **CODING AND COMPOSITION – Power in the university**

If the experience of coding and composing aren't alien endeavours, if they share some things in common with respect to the process of organizing thought, with respect to the tools they use, and the how one actually scribes, the activities also intersect with respect to the relationship between code, writing and academic power.

In recent years, we've seen how words can subtly change the political landscape and change how people think about the world around them. By calling "global warming" "climate change" and by calling the "estate tax" the "death tax," politicians have been able to reframe many people's political perspectives. Programmers aren't rhetoricians, so they can't change people's minds in this same way. And since they don't rank very high in the academic hierarchy, they don't have the clout or power to coerce people into doing things when they can't be persuaded through the use of words.

But that doesn't mean that coders aren't wielding power in more insidious ways. Coding appears benign because it doesn't appear overtly coercive. But programmers play a pivotal role in defining the university's virtual worlds – and these days, virtual worlds are taking up increasing space in the academic landscape. The power of the coder would be trivial if we could simply turn off our computers and walk away from them. But every year we spend more and more time online. As our worlds become more virtual, the power of the people who define these worlds increases. The apocryphal stereotype of the meek and reclusive programmer, the person who has not a single iota of charismatic power is an irony because the programmer stands a decent chance of embedding her own ideas of how the university should work in her programs while the rest of the university is bickering about written policies.

Here, for example are some ways that code channels power: For how

many years have your colleagues complained about a particular computer system on campus that they are forced to use? Perhaps it's your slow-as-snails mail system, or a cumbersome online gradebook, or the new but completely labyrinthine system that Human Resources makes you use in order to hire new associates. If these were bureaucratic processes that were written in your university's Policies and Procedures Manual, then people who have been vested with traditional authority could use traditional forms of power to change these rules. But increasingly programmers code these rules (or at least the more miscellaneous rules) in places far removed from the public forums in which these rules have traditionally been written.

People might object to these rules, but by the time they get a chance to object often they've already been codified and presented to them as *fait-accomplis*. Walter Ong, in attempting to describe how writing is inherently more autonomous and contumacious than orality, has said much the same about writing:

Writing establishes what has been called 'context-free' language or 'autonomous' discourse, discourse which cannot be directly questioned or contested as oral speech can be because written discourse has been detached from its author....The author might be challenged if only he or she could be reached, but the author cannot be reached in any book. [Ong, p.78-79]

For Ong, writing enjoys a type of autonomy and power that speech doesn't have. The same can be said about computer code, only to a greater degree. Code describes the world in a certain way, and the world is forced to conform to its reality whether it wants to or not. For Ong, writing "technologized" the spoken word, and vested its authors with unique and expanded powers. Code continues this technologization of the word and, arguably, vests expanded powers in the coder. To boot, unlike writing, code doesn't need a willing executive to carry out its vision – the machine and the code work to carry out the bidding of the programmer even after the programmer has left town.

Of course there are limits to this power -- coders typically don't make the types of decisions that are vested in an executive. Still they often wield more power than petty bureaucrats. Coders manipulate the rules from behind the scenes and late at night, if they code prolifically they can change a lot of rules, and their code can operate relatively autonomously without a human agent. In its more traditional and recognizable forms, power flows downward in the university from trustees, to president, to provost to faculty and eventually to the

custodians and coders. The president's or faculty senate's fiat, as codified in a new written policy, legitimates and channels power. But as IT grows, and the presence of code grows, power takes on a new and more insidious character. Power may continue to flow from the top down in written edicts but it also seeps in from the bottom and the side as more and more of academic life is framed and defined by code.

Time and again technology critics comment on the "unintended consequences" of technology. But while this is a known effect, part of the reason it's unanticipated is because we think of information technology (and programmers) as relatively benign. Our attention is drawn to the power that emanates from above rather than from the more insidious forces that reshape and change our lives from the sides and from below. The culture of academe encourages this blindness.

Although columnists like Thomas Friedman have attributed real power to the technicians who have flattened the world and turned India and China into contenders for world economic power, in academia proper the formal power of the technician is barely acknowledged. In their own house faculty understandably want to continue believing that technology is still there to serve them and that it's an instrumental rather essential function of university life. Canonical texts in academe lend credence to this vision. Aristotle, Arendt, and Allan Bloom in their various intonations have forwarded this vision. The life devoted to reproducing the material goods needed for living from one day to the next are subordinated to the life that is devoted to articulating thoughts through philosophy or overt political speech. And to the extent that academics see the university as an embodiment of an Aristotelian or Arendtian utopia, the role and power of the technician will be underestimated. But as I have tried to show, programmers are not ordinary technicians. Like the philosophers and politicians which Aristotle and Arendt (and which much of the academy) hold in highest esteem, they too are capable of scribing thought into written words.

And while these words aren't written in the same languages that Aristotle and Arendt envisioned would act as a vehicle of power, code carries a power that is as influential and insidious as it is underestimated.

## **CONCLUSION**

Computer languages and written languages differ in significant ways.

Among other things, written languages that are also spoken express and delineate emotional experiences more expeditiously than a typical computer language. They expand the consciousness of their users in radically different ways. The type of knowledge that is delineated and

contained by a spoken or written language is different than the type of knowledge that is made available through computer code. But these differences, which serve to legitimate a divide between the culture of composers and the culture of coders, also serves to mask remarkable similarities between these two cultures. The experience of composing prose and composing code resonate in ways that deserve note. Both sets of practitioners are using the tools of writing or coding to hone their use of these tools. This honing is what is sometimes called technique, and the techniques of writers and coders are not all that dissimilar. Both cultures systematize their techniques through the use of advanced editing tools and (increasingly) through the use of advanced collaboration tools. Both cultures are confronted with the challenges of imagining one's audience and negotiating transitions from the quiescent spaces in which code and composition is produced to the more social spaces in which code and composition is refined and consumed. Finally, in the university at least, code and composition (whether in a computer program, a PPM or in the delineation of politically correct discourse) have discrete and concrete effects on the way that power is distributed and wielded in academe.

Departmentalization, the college catalog, and the differences between spoken/written languages and coding languages serve to cleave the activity of composition and the activity of coding into separate cultures in ways that resonate with C.P. Snow's complaints of 40 years ago.

But if we can look past these differences, many more commonalities exist than some people would initially presume. Coding doesn't expand one's emotional universe as readily as writing. But this doesn't mean that the activities are alien – in fact they should be familiar to one another. Programmers (like writers) are attempting to create order and meaning out of disorder. Programmers (like writers) also have to negotiate transitions between social worlds and more quiescent worlds where the code is actually generated. And programmers (like writers) also scribe as a way of channeling power.

Writers and programmers might not have occasion to intersect much as they make their separate ways across campus. But coders and composers share a common set of tools, techniques and challenges in their quest to fashion order out of disorder.

By Luke Fernandez, Ph.D., Weber State University, Ogden UT; his e-mail address is lfernandez@weber.edu. Fernandez's publications have appeared in numerous periodicals, including the Chronicle of Higher Education, Peer Review Magazine, and SignPost, among others. He is Assistant Manager of Program and Technology Development at Weber State.

