Computers and Composition 16, 325–339 (1999) ISSN 8755-4615
© 1999 Elsevier Science Inc. All rights of reproduction in any form reserved.

The Politics of the Code

JOEL HAEFNER

Illinois Wesleyan University

This article explores the software behind the interface of the programs we use in composition classrooms—the principles of Structured Programming (SP), file structure, and some other significant issues in programming—and places the production of software in a cultural context. Programming protocol is shaped by two major strands: the profit imperative and the hierarchical structure of corporate America. The assumptions and procedures of Structured Programming are critiqued, and hypertext, structured programming, and natural-language writing are compared. This article suggests that writing instructors think about ways to customize programs used in their composition classes and to understand the implications and limitations of the necromancy of software.

ANSI C language coding file structure hypertext programming scripting Structured Programming

WRITING ON THE WORLD

My article title alludes to—better yet, engages with—Cynthia Selfe and Richard Selfe's (1994) important *College Composition and Communication* article, "The Politics of the Interface." Selfe and Selfe acutely analyzed the cultural metaphors and premises underlying some of the common computer interfaces composition instructors and students work with everyday, particularly the Western (further: American), English-speaking, and capitalistic iconography that controls the two most popular interfaces, the Macintosh and Microsoft WINDOWS operating systems. The goal of Selfe and Selfe (1994) was "to help teachers identify some of the effects of domination and colonialism associated with computer use so that they can establish a new discursive territory within which to understand the relationships between technology and education" (p. 482).

This piece continues that quest. What I intend to do here is move this analysis a step back, back into the language that writes the interface, the principles of Structured Programming (SP), file structure, and some other significant issues in programming. Like Selfe and Selfe (1994), I find that programming protocol is shaped by two major strands: the profiteering imperative and the hierarchical structure of corporate America. The ubiquity of American code, of American interfaces and operating systems, and of American hardware, is a new kind of cultural imperialism. As Selfe and Selfe (1994) pointed out:

Direct all correspondence to: Joel Haefner, PO Box 2900, Illinois Wesleyan University, Bloomington, IL 61702-2900. Email: <jhaefner@sun.iwu.edu>.

> Within the virtual space represented by these interfaces, and elsewhere within computer systems, the values of our culture—ideological, political, economic, educational—are mapped both implicitly and explicitly, constituting a complex set of material relations among culture, technology, and technology users. (p. 485)

It is instructive, on this same point, to read drafts of the American National Standards Institute (ANSI) standard for programming languages, such as the very popular and powerful C language. As the X3J11 committee struggled with its draft standard, the International Standards Organization (ISO) demanded that ANSI address the issue of international use of C language: "Some features were introduced to give C a more international scope. These include multibyte characters, locales, and some specialized library functions" (*ANSI C*, 1988, p. 3). But, despite a special function ("setlocale") that allows C to "speak the local dialect, as best you can," American English (with its 26 characters, no diacriticals) remained "the lowest common denominator" (Plum, 1987, p. 80). This amounts to cultural colonialism and "discursive privilege," as Selfe and Selfe (1994) pointed out; a change to a more universally accessible and acceptable base language, like Unicode, would "work against a complex set of tendential forces encouraging inertia" (p. 491).

On the surface, most instructors aren't concerned with computer code and programming—it doesn't really seem to impact the classroom. But I want to suggest some good reasons for thinking about the nature and structure of programming: the options we have as instructors and students for customizing the software we use, the fact that more of us will be "programming" (or, more properly, scripting) with our students on the Web, and the broad pedagogical and epistemological implications of programming structures.

Certainly, there are continuities and discontinuities between the writing that goes on in our composition classrooms and the writing of code. In one sense, computer code is ultimately (and much more so than exposition) action: you write, you run it on the computer, and it works or bombs. Software applications perform "a kind of writing on the world," as Jay David Bolter (1991) styled it (p. 10). Verbal representation is immediately turned into an electronic event. Using the terminology developed by J. L. Austin (1961) and John Searle (1969), computer code is clearly *performative* language. Austin and Searle would probably both call code statements *illocutionary acts* (Austin, 1961, p. 149; Searle, 1969, pp. 23–24), but it is important to note that this is one of only four performative verbal actions that Searle identified. In other words, the language of coding is much more limited in the kinds of actions it can perform than the language we use for communication every day, and in writing classes. And, this makes sense; a programming language runs the computer and manipulates data. Verbal language can persuade, assert, refer, inform, question—the list goes on and on; language has a spectrum of tasks in a cultural universe much broader than the virtual world of microcomputers, mainframes, and even global networks. Bolter (1991) pointed out that computer language is formal language, not natural language:

> Programming languages (like PASCAL or C) constitute a restricted and yet powerful mode of communication, a mode based on imperative sentences and the unambiguous use of symbols. Admittedly, their rigid syntax makes these computer languages unusual; natural language is far less precise. And unlike natural language, computer language is made to be written down: it belongs on the page or the computer screen. (p. 9)

As Selfe and Selfe (1994) noted, quoting Turkle and Papert, formal, logical thinking—Boolean algebra, really—has become canonized in computer science, carrying with it implications of positivism and hierarchical cognition.

Yet, programming languages aspire to be more natural and less cumbersome, more like spoken American English. Basically, there are three levels of code: machine code, written entirely in binary numbers; assembler code, which includes some English syntax but still uses binary locations for operations and data storage; and "high-level" languages, like C or PASCAL, which rely heavily on the assembler, use only English-like terms and syntax, do not deal with binary memory locations, and are somewhat automated but still require precise syntax and definition. A new generation of languages "event-driven" and written for graphical interfaces like Microsoft WINDOWS or the Macintosh OS, automatically correct or question syntactical mistakes and are much more flexible in terms of defining variables, creating interfaces, etc.[1] Presumably at some point in the future, as programming languages become even more sophisticated, natural language and code will converge even more, and, in fact we are seeing this convergence in software that automatically converts text (e.g., WORDPERFECT for the PC and Microsoft WORD) into the HTML (hypertext markup language) needed to create Web pages.

Bolter (1991) pointed out that although there are many clear differences between writing code, let us say, and writing a personal essay, there are similarities as well, and computer code leads one to "understand natural language too as a network of interconnecting signs" (p. 10). To some extent, code and natural language converge or diverge depending on how one situates oneself within the field of language and textual criticism: Is code denotative and formalistic, or is it connotative and intertextual? Bolter (1991) and George Landow (1992) argued that hypertext bears strong affinities with poststructuralists like Barthes, Derrida, and Bakhtin.

In fact, traditional code, designed along the lines of Structured Programming, is quite different from hypertext. Although the coding behind hypertext (traditionally called scripting) is indeed coding, it is much more flexible: It is event- and user-driven, presented within a graphical interface, highly automated with point-and-click protocols, and designed to link data within an ongoing process, not manipulate data in a product-driven environment. Traditional code resembles what Foucault (1972) identified as the *binary* nature of post-Renaissance language:

> From the seventeenth century . . . the arrangement of signs was to become binary, since it was to be defined . . . as the connection of a significant and a signified. . . . [After the Renaissance language was] fixed in a binary form . . . and . . . language, instead of existing as the material

---

[1]Object-oriented Programming (OOP) represents an approach between structured programming and scripting. SMALLTALK, the prototype for object-oriented languages, was developed at Xerox PARC and was designed for a single computer. Later, object-oriented languages, developed during the 1980s, were event-driven and GUI-based. However, many writers have pointed out the similarities between sequential and hierarchical programming strategies and OOP, as well as conceptual interstices such as type and class, and variable and object. Although OOP is beyond the scope of this article, some key ideas developed in SMALLTALK and later in object-oriented languages—specifically encapsulation, inheritance, and polymorphism—enhance the portability, reusability, and accessibility of OOP, which may dovetail in interesting ways with collaborative writing theory. (See Beaudouin-Lafon, 1994; Budd, 1991; Graham, 1994.)

> writing of things, was to find its area of being restricted to the general representation of representative signs.[2] (p. 42)

Indeed computer code seems to epitomize this kind of language; it is binary in two senses: in the sense that every word or command is (at some deep level) translated into binary machine code, and in the sense that code relies on Boolean logic, a chain of alternative choices. And the syntax, definitions, data structures and command structures of programming languages are profoundly self-referential: They represent statements and actions within the computer's combinatory circuitry, not things outside the box. In short, code—and the post-Renaissance binary language Foucault described—represents a virtual reality. Clearly, Foucault laments this alteration; the "prose of the world" has been replaced by the discourse of signification; only literature, a "counter-discourse," revives "the profound kinship of language with the world" (p. 43).

The dichotomy between formal/logical/binary programming languages and connotative/referential natural language I have been discussing here, and which is expressed as well by Bolter (1991), Landow (1992), Selfe and Selfe (1994), and Theodor H. Nelson (1987), falls into the very trap of binarism that Foucault (1972) claimed has afflicted Western culture for centuries. It may well be that the textual and institutional boundaries of academic/analytical discourse demand this kind of cognitive binarism. Let's turn to a specific example to illustrate and contextualize this binarism.

Here is a small, functioning code written in C, a translation of one of the most famous conditional statements in English literature:

```
#include <stdio.h>
int
main(void)
{
char tobe;
printf("\nTo Be?\n\n");
printf("Enter Y for Yes, N for No ");
tobe = getche();
tobe = toupper(tobe);
if (tobe == 'Y'){
printf("\n\n'tis nobler to suffer the slings and arrows of outrageous fortune\n");
}else{if (tobe != 'Y'){
printf("\ntake arms against a sea of troubles\n");}
}
getch();
clrscr();
return(0);
}
```

And the natural language version, compliments of William Shakespeare:

---

[2]Foucault (1972) discussed a similar division in discourse, the modern period being preoccupied with language as a manifestation of "the will to truth"—that is, language as a representation of what is true or false, not language as a reflection of the material world or as a vechicle of political power (pp. 216–220). Language became self-referential, rule-governed, and hierarchical after the Renaissance, Foucault claimed. It is not difficult to see computer code, based as it is on Boolean logic and referring exclusively to the electrobinary world of the computer itself, as a pure expression of the kind of language Foucault is talking about here.

> To be, or not to be—that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune Or to take arms against a sea of troubles And by opposing end them.

Of course this is contrived, but it is useful in pointing out some of the differences between code and poetry. First, notice in the code the line *char tobe*. This is a variable definition; in other words, the data type of the variable must be precisely defined (*char*, or character) in the correct syntax (note the semi-colon at the end of the line). We know *tobe* is a character; its value will be assigned as the program runs. In Shakespeare's version, exactly what "to be" means is open to interpretation—it may well mean a choice between life and death, but not necessarily. Second, the code offers the reader or user a stark choice: if *tobe* is true, the computer prints the line about outrageous fortune on the screen; if *tobe* is false, "take arms against a sea of troubles" appears on the screen. Only two alternatives are possible. The simultaneous dichotomy Shakespeare demands—to consider being and nonbeing—cannot exist in the text of code. Third, the entire allusive and metaphoric dimensions of the blank verse disappear; code does not allow multiple definitions of variable names, and the use of reserved words (e.g., "if") is strictly limited. Fourth, Hamlet's soliloquy from the first scene of the third act goes on for 28 more lines, complicating the choice between being and nonbeing, adding images, examples, and references. This might constitute multiple data entry, a cardinal sin in Structured Programming. Fifth, by line 83 of the soliloquy, fear of the afterlife has been linked to inertia and inaction—an associative, not a logical connection. Finally, the repetition of "To die, to sleep" throughout the passage, which gives it much of its incantatory power, violates the linear, hierarchical, and efficient (read: nonrepetitive) rules of coding languages.

Although Hamlet's lines are a meditation on inaction and paralysis in the dynamic world of Shakespeare's Denmark, the small C code above is an imperative of computer action. Structured code is never so "sicklied o'er with the pale cast of thought" as to "lose the name of action." As the code performs in the computer, bursts of electricity trip circuits; as Hamlet speaks on the stage, words flow, complicate and negate themselves, but no action takes place. Structured programming always chooses "to be"; Hamlet dangles between the two.

THE NATURE OF STRUCTURED PROGRAMMING

Structured Programming (SP) had very humble origins: In 1965, Edsger W. Dijkstra, of the University of Eindhoven in the Netherlands, suggested at the IFIP Congress that the GOTO statement could be excised from programming languages and hence from programs (Dijkstra, 1965). The GOTO statement—familiar to anyone who has written a simple batch file in DOS, or who has played around with the AUTOEXEC.BAT file controlling their PC—simply allows the programmer to jump around to different subsections, marked with a line called a label, inside the code. The GOTO statement was a mainstay in early versions of FORTRAN, but following the SP revolution the GOTO statement became a pariah in the family of computer statements and languages (Yourdon, 1975). In a famous letter to the editor of *Communications of the ACM* in March 1968 (Dijkstra, 1968a), in an article in the same journal two months later (Dijkstra, 1968b), and in a collection the following year (Dijkstra, 1969), Edsger Wybe Dijkstra continued to call

for the elimination of the GOTO statement and to promote top-down design, an essential feature of Structured Programming.

One of the most widely disseminated versions of Structured Programming was Edward Yourdon and Larry L. Constantine's (1978) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* and Yourndon's (1975) *Techniques of Program Structure and Design*. Yourdon's (1975) general definition of SP is quite concise: "Structured programming is a philosophy of writing programs according to a set of rigid rules in order to decrease testing problems, increase productivity, and increase the readability of the resulting program" (p. 144). As it evolved in the 1970s, SP included these salient points:

- **Top-down design.** The most general solution or algorithm needs to be written first, with subsequent coding to follow.
- **Modularity.** In other words, the program should be broken down into independent subunits; those modules can be used repeatedly throughout the main program.
- **Sequentiality.** A program should run in sequence through the main code and the modules it calls. Jumping around within code (using GOTO statements) is poor practice.
- **Limited routines.** Just two "structures" or procedures are needed in SP: routines that offer binary choices or alternatives (like *if-then* statements) and routines that repeat actions until a certain threshold is reached (like *do-while* statements).

There are important premises underlying each of these features of SP, and it's worthwhile to analyze, briefly, some of these premises.

Although the specific origin of SP was Dijkstra's attack on GOTO statements, the cultural imperative catalyzing SP has always been corporate productivity and profitability. Yourdon (1975), for example, devoted separate units in his section on "What are the Qualities of a Good Program?" to "Minimize Testing Costs," "Minimize Development Costs," and "Minimize Maintenance Costs." "We are interested in systems that are cheap to develop, cheap to operate, cheap to maintain, and cheap to modify," Yourdon wrote elsewhere (Yourdon & Constatine, 1978, p. 12). In fact, the whole philosophy of SP was grounded on corporate structure:

> One of the fascinating aspects of program design is its relationship with human organizational structures—particularly, the management hierarchy found in most large corporations. Whenever we wish to illustrate a particular point about program design . . . we often can do so by drawing analogies with a management situation. (pp. 22–23)

In other words, the basic premise behind most computer programs is profitability. And, the very structure of most programs—with a controlling main algorithm (top-down design), modules that work in isolation and don't communicate directly with each other, a rigid sequential protocol, and a limited repertoire of strategies for getting things done—mirrors the American corporate hierarchy. Figure 1 is a structure chart of a simple C language program. Compare that to Figure 2—the management structure of the engineers division of Monsanto in the late 1960s.

When I showed these diagrams to a colleague, he commented that they "bordered on plagiarism." In fact, systems analysts are well versed in corporate and project hierarchy

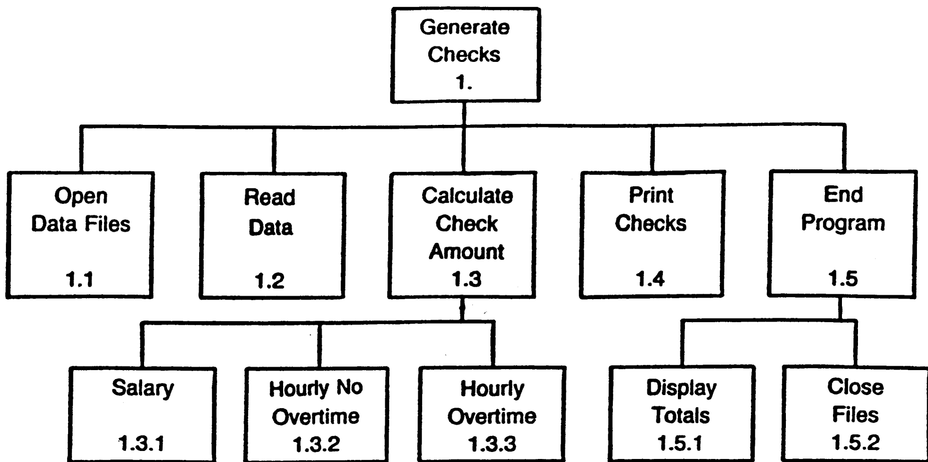

Figure 1. Structure chart of a simple C language program (Bloom, 1987)

structures and apply those same organizational strategies to structuring the computer programs that they oversee.

The principles of Structured Programming, top-down design, and the whole hegemony of modern programming practices are articulated by the IEEE (Institute of Electrical and Electronic Engineers) Standards Board. In a series of monographs outlining software standards throughout the 1980s, the IEEE entrenched industry-wide rules for software management, development, and testing (see IEEE, 1983, 1984, 1988a, 1988b). Even within the field of business programming, some writers realize that software standards are counterproductive. Carl F. Cargill (1988), from the Digital Equipment Corporation, for example, asserted that the whole standard-creating process is grounded in the scientific method (because engineers sit on the standards committees), although standards for software development ought to be based more on marketing factors. The principles of Structured Programming and the rules propagated by the IEEE continue to be widely taught in high school and college-level programming classes, as Shmuel Rotenstreich (1988) has documented and advocated.

The principles of Structured Programming are summarized as: top-down design, modularity, sequentiality, and limited routines (or control structures) stem from, and demand, hierarchical, teleological, and atomistic cognitive strategies. The goals of SP are to create a main algorithm that will control much smaller units, the modules, in what is sometimes called a master-slave relationship (a term also used in networking lexicon); to create an algorithm that moves, linearly, toward a clear solution to a clearly expressed problem; to break the problem into the smallest possible units so they can be reused as needed in the program, easily coded, and easily tested; and to limit the number of possible logical routines or structures so coding doesn't become too confused and intimidating. These principles have proven very successful in computer programming, and any programmer will tell you that modularity and limited routines simplify and even make possible complicated coding tasks. I add that in reality Structured Programming is probably less rigidly adhered to than one might think. Programmers work collaboratively

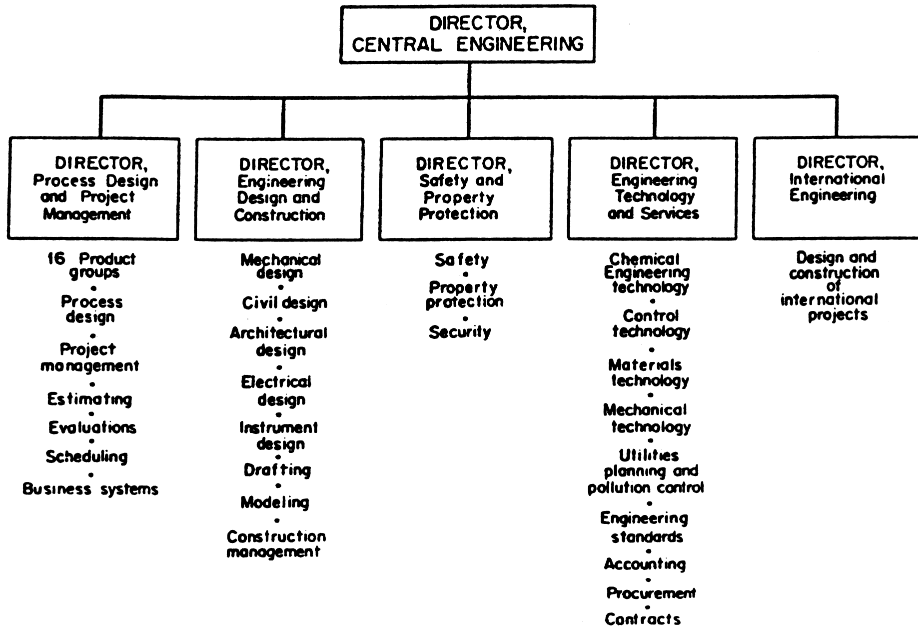

Figure 2. The management structure of the engineers division of Monsanto in the late 1960s (Holzapfel, 1969)

in teams, and the writers of a large code base can change over time, with different styles and different logical strategies (even with limited routines) for solving problems.

But it's important to note that the principles of Structured Programming aren't shared in many writing classrooms. Although portfolio-based courses seem to be very product-centered, most instructors and students would assert that it is on the process of revision and production that the pedagogical emphasis falls. The teleological/linear process of a SP code is anathema to a classroom where invention and discovery, along with multiple drafts, exploration, and in-class writing, are validated. One could, of course, easily draw a parallel between top-down design (where there is a controlling algorithm) and the thesis statement in a formal research paper; but, again, for many teachers of composition, that kind of strategy precludes a host of other varieties of writing, and other varieties of audience and situation. And, finally, both the hierarchical nature of SP and its atomistic approach may not match a student-centered, peer-reviewed pedagogy, a nonheuristic approach to invention, composing, and revising, and a holistic perspective on evaluating and commenting on writing.

The principles of SP are closely linked, naturally, to the historical development of the digital representation of data and the commercial marketing of the microcomputer. The role of IBM in such a history could not be overemphasized; many of the decisions, terminology, and strategies decreed by Big Blue in the 1960s, 1970s and early 1980s have clearly shaped the way computers work and the way code is written—and, consequently, what we have to work with in the computer-supported classroom. No one has written a more interesting or insightful statement on IBM's control of the computer industry than Nelson (1987), or of some other important companies like DEC, Xerox, and Apple.

One consequence of early decisions by IBM and other pioneers was that the way data was stored and represented led to just a few types of file structures. Most programming languages limit file types to sequential (records are accessed in sequence), binary (data is stored in binary, not textual, form), and random-access (where records can be accessed randomly, as needed, by an index number). The important point about those file structures, however, is that each file is discrete, without connections to other files. In other words, each text file students write in a computer-supported classroom stands independent of all other files. If a student writes revisions of a paper, each revision, with its duplication of text, is an independent, and independently saved, file. This fact has a number of implications for a computer-supported composition class: The file system developed by IBM is highly hierarchical (even on Macintoshes), it can be very wasteful because data stored is duplicated, and it prevents links between files within the file structure. These features work against a process-centered theory of composition that emphasizes revision and work against a highly collaborative environment where students share files and text blocks to produce communal documents.

Nelson, the "father" of hypertext, foresaw these problems thirteen years ago. In "The Tyranny of the File," Nelson (1986) noted how limiting conventional file structures were and suggested a common data bank where files would simply point to bits of data. The problem with conventional file structure, Nelson wrote, is that it dictates hierarchical thinking. "Some people think hierarchically and that's fine," Nelson declared,

> but those who don't should not be forced to do so. There are those who imagine that forcing a problem into a hierarchical structure promotes clear and rigorous thinking. But this is, to use the politest possible term, malarkey. . . . Unfortunately, the hierarchical model imposes intricate, fixed pathways that we must commit to and memorize and which are quite hard to change. . . . We should be able to work on numerous things at once, Ping-Pong style, never having to deal consciously with the formalisms of opening and closing files and applications. . . . Today's windowing packages are a start, but they just disguise underlying conventional file methods that must be grappled with as usual when the day is done. (p. 83)

Some of Nelson's ideas—especially files that simply point to other data bits—have been partially adopted in the World Wide Web and the increasingly ubiquitous hypertext file system, and this has tremendous potential for the composition classroom, as I suggest below. But, Nelson's other objections, particularly the wearisome task of opening and closing files and applications, remain a part of coding procedure and of the way we do business with our computers.[3]

CODE AND THE CLASSROOM

In practical terms, what this means is that the composition instructor teaching in a computer-supported classroom, running an industry standard like Microsoft WORD, is working with tools that either aren't necessary or aren't appropriate. Extra features of WORD include: drawing capabilities, an elaborate mail-merge feature, easily imported clip

---

[3]Nelson (1981) explained how his Xanadu Project, an early hypertext program, was designed very differently than "binary-riffle data bases of the card-catalog type" (p. 3/4), and he described there the programming protocol of hypertext, very different from Structured Programming (pp. 2/10–2/33).

art, and table-creating functions that even automatically perform columnar or row arithmetic. The average composition student doesn't need table and graphing powers, or desktop publishing capabilities, or mass correspondence features, but they're there, just the same. And, some business-centered features actually get in the way of good classroom instruction. Take the "AutoCorrect" feature, for example, which automatically will correct "teh" to "the." (To even write that sentence I had to turn off that feature, which, thank heavens, WORD will allow me to do.) Although some traditionalists might argue that students do not learn if their typing mistakes are automatically corrected, I think that argument misses the real point. The AutoCorrect module can restrict the kinds of things students can try in their writing, and sometimes it can just be hard to negotiate (as when, for example, quoting from a source that is "nonstandard" according to "AutoCorrect's" criteria). Although a feature like this can be "customized" (I have only the "smart quotes" feature activated), that customization assumes a single user for the software, not a lab or classroom environment where many students use the same software copy, the same workstation, many times a day.

A feature like "AutoCorrect" demonstrates both the limits of contemporary computer programming (a sequential search for errors from a set, albeit customizable, data bank of errors) and its roots in corporate America. The goal of that feature is simply correctness, because standard English dialect means profitable communication. Like Structured Programming itself, the tools of Microsoft WORD were based on rules and guidelines ideally suited for a business environment—not a writing classroom. One of the best examples is the "Grammar" tool. A sentence like the one that ends the preceding paragraph "was too long to analyze for grammatical structure." Long sentences either fall outside the powers of "Grammar," or they may produce a warning like the one that flagged the second to last sentence of the previous paragraph: "Consider revising. Very long sentences can be difficult to understand." "Grammar" discourages contractions, opening sentences with conjunctions, and a host of other "errors." And, even if we warn our students against using such a tool, they will be tempted to use it, and produce rule-driven writing that ignores voice and audience. And, to put it in terms of the bottom line, we're paying for something we don't want, too.

There are ways of customizing word-processing packages to fit our pedagogy, of course, and we should take advantage of those options, when possible. Needless to say, such customization isn't always possible. In a computer-supported classroom or lab, one instructor's program is another instructor's poison. Consider the "Grammar" tool for a moment. Using the "Customize" option, we could eliminate the "Grammar" and "AutoCorrect" tools from the menu bar. But if another instructor wanted one or both of those functions, how would we negotiate this? Perhaps network protocols would allow special logons accessing customized copies of word processing packages. But, already, we can see that administrative, software, and hardware costs are increasing, to say nothing of the time instructors would have to invest in customizing special versions of the software.

I mentioned Nelson's (1986) critique of traditional file structure, which is essentially proprietary: Revisions overwrite earlier versions of files, or new versions are created as separate files. Files don't share data with other files, except by incorporating that data into its own protected field. This makes some sense in a property-centered, security-anxious business environment, but it works against both collaborative and process pedagogy where we would like students to share text freely and easily track revisions in texts. To keep track

of earlier versions, students have to save multiple copies of their work; to share texts, students have to swap files, open, highlight, copy, open another file, and paste. Synchronous file-sharing isn't possible, because one version must be locked on a network. Some programs, like ASPECTS, allow a real-time version from one workstation to be shared by all workstations, but this is not the same as having all students revise a common text simultaneously.[4] The proprietary nature of files has two other logistical problems for the writing classroom: cross-platform formatting and the inefficient use of server storage. Anyone who has taught in a computer-supported environment knows the horrors of file incompatibility, and how it can eat up valuable class time. A classroom is full of Compaqs, and students use Macintosh computers in their dorm rooms—or vice versa. Students use CLARISWORKS, and you teach in WORD for Macintosh. Sure, there are software solutions for these problems. But does your department/college have the financial and personnel resources to buy and staff the software? How quickly will storage space on your classroom server be consumed as students save and review multiple revisions of their papers?

Selfe and Selfe (1994) noted that most composition teachers "deal with technology not as critics but as users," and urge teachers to become critics of technology. They go on to suggest that teachers delve into the growing body of work on composition and computers, participate in educators attempting to influence/write software, and do what they can to alter the interfaces their students face (pp. 496–500). In fact, with a great many limitations, instructors can become programmers themselves. To customize a word-processing package is, in a sense, to reprogram it, and the more recent and sophisticated programs (like WORD for WINDOWS) do have quite a few options for customizing. Take the "Grammar" tool that I derided earlier. A user can set different styles for "Grammar." Instead of "Strict" or "Business," you can choose "Casual," a plain text style that ignores contractions (but still flags opening conjunctions), or three customized settings. Like earlier word-processing software (e.g., WORDPERFECT 5.1 for DOS), a WORD user can create a custom dictionary and thesaurus. And, there are some features that can be truly useful in a writing class, such as the "Mark Revisions While Editing" option, which records all revisions to a text.

There are many other opportunities for customizing the programs we use, and we don't have to know programming languages to exploit them. This is one of the alternatives Selfe and Selfe (1994) mentioned, and critiques of new software specially designed for writing classes and reviews of more mainstream software packages regularly appear in the pages of *Computers and Composition*. The most active field for instructors interested in taking some control of programming, and having their students take control of code, remains hypertext, and even more importantly, HTML on the World Wide Web. This is not the place to dive into an analysis of hypertext and HTML. But in terms of the issues this piece has raised—the binarism and hierarchical nature of Structured Programming and its implications for the classroom—hypertext is a significant exception. I'm tempted to call the Web "Theodor Nelson's Revenge," because many of the features of traditional programming he deplored (exclusion of the GOTO statement, linearity, proprietary file structures) are circumvented with hypertext. Three important pedagogical events happen in hypertext: the authoritarian and proprietary status of the author/programmer is weak-

[4]Janis Forman (1991) provided a good overview of some of the more important theoretical and research issues.

ened;[5] the active participation/programming of the user/reader is empowered;[6] and the text itself becomes much more fluid, much more open to collaborative intervention.[7] The relatively easy-to-understand language of HTML or stand-alone programs like HYPERCARD or GUIDE makes it feasible for instructors to learn and teach this kind of scripting or programming in their writing classes. Certainly, many are doing so now, at least many are at my institution.

There are many perils involved in teaching coding, even hypertext scripting, in a composition classroom. Anyone scripting remains limited by the syntax and functions of the language and by the degree to which the original programmer allowed access and customizing by users. There are technical hurdles: student and instructor resistance and frustration, institutional inertia, and the simple questions, is this writing and does it belong in a writing class? Even if one answers those global questions and forges ahead with hypertext or HTML in class, a host of other pedagogical and theoretical issues arise; for example, what happens to our concept of voice or audience? (see Emily Golson, 1995). As Johnson-Eilola (1994) noted, “hypertext is useful as a measure of the inadequacies of current pedagogy” (p. 203).

And, even more importantly, how does the formality, the action-directedness of coding, and even hypertext scripting, affect our and students’ notions of writing and reading? What becomes of our sense of reading and writing for meaning and structure when a hypertext or a reading of the Web can always be different, can always take different paths?[8] What happens to our sense of the subjectivity and sincerity of writing when other

---

[5]Landow (1992), in fact, explicitly made the connection to Barthe’s famous conceit of “The Death of the Author,” but almost every author writing on hypertext makes the same point: The traditional authority of the writer is undermined because the reader takes a more active and creative role in the construction of the text. For example, Bolter (1991) argued “the author is no longer an intimidating figure, not a prophet or a Mosaic legislator in Shelley’s sense” (p. 153), and Johndan Johnson-Eilola (1994) added that “in constructive hypertexts. . . the original author-text’s authority begins to evaporate. . .” (p. 207). Richard Lanham (1992) also addressed the legal issue of proprietary rights with electronic texts. (See also the solid overview of intellectual property issues and hypertext in Haynes, 1991.)

[6]For Lanham (1993), the rhetorical elements of style and content become *user-definable* in electronic texts (especially hypertexts): “. . .the figure of the hypertext author approaches, even if it does not entirely merge with, that of the reader; the functions of reader and writer become more deeply entwined with each other than ever before” (p. 71).

[7]On one hand, many of the commentators on hypertext envision the media as a direct attack on traditional textuality, like Charney’s (1994) claim that “Hypertext has the potential to change fundamentally . . . how we conceive of text itself” (p. 239). On the other hand, even enthusiasts for hypertexts like Moulthrop and Kaplan (1994) pointed out that “Hypertext represents an evolutionary outgrowth of late-modern textuality” (p. 221). Lanham (1993) celebrated most unreservedly the “fundamental irresolution” of the digital text (p. 7).

[8]Although two recent studies have shown that hypertext mapping functions “had no effect on recall, comprehension, or recall of text structure” (Wenger & Payne, 1994), such spatial configurations will be necessary for analysis of a hypertext. Lanham (1993) is very prescient here, suggesting that future rhetorical theory will not be able to eschew “visual thinking,” but will have to deal with spatial and iconic expression *as well as* verbal text; Lanham quoted Susanne Langer on the simultaneous, not successive, nature of visual forms (p. 77). Smith (1994, pp. 270–276) also applied Langer’s idea of

texts and voices invade a text with a click on a link? As Henrietta Nickels Shirk (1994) wrote, "the challenges connected with hypertext communication are not so much technological as philosophical and conceptual" (p. 199).

Shirk (1999) is right, of course, that hypertext does present philosophical and conceptual questions—but, in fact, any coding system does. Nelson (1981), in his autobiographical exposition of hypertext, *Literary Machines*, said that hypertext grew out of his interest in "the structure of *ideas*, and . . . how to set computers up to hold them" (p. 1/20). How programmers go about writing code structures the ways we (can) use computers, and hence it structures the ways we think about ideas and how to manipulate them. The hierarchical, binary, and sequential form of Structured Programming and its grounding in the American corporate environment have profound implications for the kinds of writing students do in computer-supported classrooms and the ways we teach that writing. "A computer language is a system for casting spells," Nelson (1986) wrote, "based in part on the personality and preoccupations of the person or people who designed it" (pp. 56–57). If that is true, then English instructors in computer-supported classrooms need to know something about the context and the necromancers of the code.

Joel Haefner is the Writing Coordinator at Illinois Wesleyan University and is completing a Master's degree in Computer Science. A PhD from the University of Iowa, he has published articles in *College English*, *English Journal*, *College Literature*, *Prose Studies*, and other journals. His 1994 coedited volume, *Re-Visioning Romanticism*, was a Choice Outstanding Academic Selection.

REFERENCES

ANSI C. (1988). *A lexical guide*. Englewood Cliffs, NJ: Prentice Hall.

Austin, J. L. (1962). *How to do things with words*. Oxford: Oxford University Press.

Bernstein, Mark. (1991). The navigation problem reconsidered. In Emily Berk & Joseph Devlin (Eds.), *Hypertext/hypermedia handbook* (pp. 285–298). New York: McGraw-Hill.

Bloom, Eric P. (1987). *The C trilogy: A complete library for C programmers*. Blue Ridge Summit, PA: TAB Books.

Bolter, Jay David. (1991). *Writing space: The computer, hypertext, and the history of writing*. Hillsdale, NJ: Lawrence Erlbaum.

Cargill, Carl F. (1988). A modest proposal for business based standards. *Proceedings of the Computer Standards Conference 1988* (pp. 60–64). Washington, DC: Computer Society Press of the IEEE.

Charney, Davida. (1994). The effect of hypertext on processes of reading and writing. In Cynthia L. Selfe & Susan Hilligoss (Eds.), *Literacy and computers: The complications of teaching and learning with technology* (pp. 238–263). New York: MLA.

Dijkstra, Edsger W. (1965). Programming considered as a human activity. *Proceedings of IFIP Congress, 65* (pp. 112–121). Washington, DC: Spartan Books.

---

thick cognition and Walter Kintsch's theories of discourse analysis to suggest that we need a fuller understanding of how hypertext reading/creating takes place: a complex, messy mapping of nodes and links followed by an integrative process that validates some links over others. Two technical articles that deal with navigation in hypertext are Gay and Mazur (1991), and Bernstein (1991).

Dijkstra, Edsger W. (1968a). Go-to statement considered harmful. *Communications of the ACM, 11,* 147–148.

Dijkstra, Edsger W. (1968b). The structure of the THE-Multiprogramming System. *Communications of the ACM, 11*, 341–346.

Dijkstra, Edsger W. (1969). Structured Programming. In *Software engineering techniques* (pp. 84–88). Brussels: NATO Scientific Affairs Division.

Forman, Janis. (1991). Computing and collaborative writing. In Gail E. Hawisher & Cynthia L. Selfe (Eds.), *Evolving perspectives on computers and composition studies: Questions for the 1990s* (pp. 65–83). Urbana, IL: NCTE.

Foucault, Michel. (1972). *The archaeology of knowledge and the discourse on language*. (A. M. Sheridan Smith, Trans.). New York: Pantheon.

Gay, Geri, & Mazur, Joan. (1991). Navigating in hypermedia. In Emily Berk & Joseph Devlin (Eds.), *Hypertext/hypermedia handbook* (pp. 271–284). New York: McGraw-Hill.

Golson, Emily. (1995). Student hypertexts: The perils and promises of paths not taken. *Computers and Composition, 12*, 295–308.

Haynes, Stephen L. (1991). Intellectual property and licensing concerns. In Emily Berk & Joseph Devlin (Eds.), *Hypertext/hypermedia handbook* (pp. 227–242). New York: McGraw-Hill.

Holzapfel, F. J. (1969). Multiple ladders in an engineering department. In David I. Cleland & William R. King (Eds.), *Systems, organizations, analysis, management: A book of readings* (pp. 303–307). New York: McGraw-Hill.

IEEE. (1983). *IEEE standard for software configuration management plans*. New York: Institute of Electrical and Electronic Engineers.

IEEE. (1984). *IEEE standard for software quality assurance plans*. New York: Institute of Electrical and Electronic Engineers.

IEEE. (1988a). *Proceedings of the computer standards conference 1988*. Washington, DC: Computer Society Press of the IEEE.

IEEE. (1988b). *IEEE standard for software project management plans*. New York: Institute of Electrical and Electronic Engineers.

Landow, George P. (1992). *Hypertext: The convergence of contemporary critical theory and technology*. Baltimore, MD: Johns Hopkins University Press.

Lanham, Richard A. (1992). Digital rhetoric: Theory, practice, and property. In Myron C. Tuman (Ed.), *Literacy online: The promise (and peril) of reading and writing with computers* (pp. 221–244). Pittsburgh, PA: University of Pittsburgh Press.

Lanham, Richard A. (1993). *The electronic word: Democracy, technology, and the arts*. Chicago, IL: University of Chicago Press.

Johnson-Eilola, Johndan. (1994). Reading and writing in hypertext: Vertigo and euphoria. In Cynthia L. Selfe & Susan Hilligoss (Eds.), *Literacy and computers: The complications of teaching and learning with technology* (pp. 195–219). New York: MLA.

Moulthrop, Stuart, & Kaplan, Nancy. (1994). They became what they beheld: The futility of resistance in the space of electronic writing. In Cynthia L. Selfe & Susan Hilligoss (Eds.), *Literacy and computers: The complications of teaching and learning with technology* (pp. 220–237). New York: MLA.

Nelson, Theodor H. (1981). *Literary machines: The report on, and of, project Xanadu concerning word processing, electronic publishing, hypertext, thinkertoys, tomorrow's intellectual revolution, and certain other topics including knowledge, education and freedom* (3rd ed.). Privately printed.

Nelson, Theodor H. (1986). The tyranny of the file. *Datamation, 15*, 83–86.

Nelson, Theodor H. (1987). *Computer lib/dream machines* (Rev. ed.). Redmond, WA: Microsoft Press.

Plum, Thomas. (1987). *Notes on the draft C standard*. Cardiff, NJ: Plum Hall.

Rotenstreich, Shmuel. (1988). The study of programming standards in computer science programming courses. *Proceedings of the Computer Standards Conference 1988*, (pp. 80–82). Washington, DC: Computer Society Press of the IEEE.

Searle, John R. (1969). *Speech acts: An essay in the philosophy of language*. Cambridge, MA: Cambridge University Press.

Selfe, Cynthia L., & Selfe, Richard, Jr. (1994). The politics of the interface: Power and its exercise in electronic contact zones. *College Composition and Communication, 45*, 480–504.

Shirk, Henrietta Nickels. (1994). Hypertext and composition studies. In Cynthia L. Selfe & Susan Hilligoss (Eds.), *Literacy and computers: The complications of teaching and learning with technology* (pp. 177–202). New York: MLA.

Smith, Catherine F. (1994). Hypertextual thinking. In Cynthia L. Selfe & Susan Hilligoss (Eds.), *Literacy and computers: The complications of teaching and learning with technology* (pp. 264–281). New York: MLA.

Wenger, Michael J., & Payne, David G. (1994). Effects of a graphical browser on readers' efficiency in reading hypertext. *Technical Communication, 41*, 224–233.

Yourdon, Edward. (1975). *Techniques of program structure and design*. Englewood Cliffs, NJ: Prentice-Hall.

Yourdon, Edward, & Constantine, Larry L. (1978). *Structured design: Fundamentals of a discipline of computer program and systems design* (2nd ed.). New York: Yourdon Press.