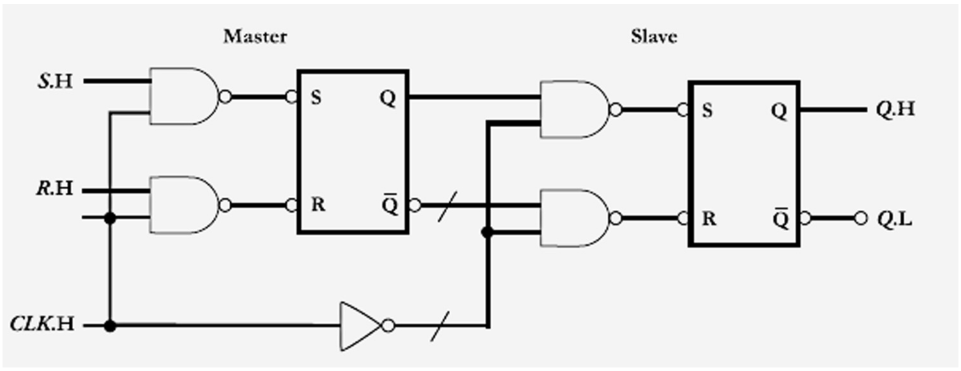


Master-slave clocked RS flip-flop.

Final version published in Grey Room 18, Winter 2004, pp. 26–51.

On Software, or the Persistence of Visual Knowledge

WENDY HUI KYONG CHUN

When enough seemingly insignificant data is analyzed against billions of data elements, the invisible becomes visible.
—Seisint[1]

Jean Baudrillard in *The Ecstasy of Communication* argues "*we no longer partake of the drama of alienation, but are in the ecstasy of communication.* And this ecstasy is obscene" because "in the raw and inexorable light of information" everything is "immediately transparent, visible, exposed."[2] Although extreme, Baudrillard's conflation of information (and thus computation) with transparency resonates widely in popular and scholarly circles, from fears over and propaganda behind national databases to examinations of "surveillance society." This conflation is remarkably at odds with the actual operations of computation: for computers to become transparency machines, the fact that they compute—that they *generate* text and images rather than merely represent or reproduce what exists elsewhere—must be forgotten. Even when attached to glass tubes, computers do not simply allow one to see what is on the other side but rather use glass to send and receive light pulses necessary to re-create the referent (if one exists). The current prominence of transparency in product design and political and scholarly discourse is a compensatory gesture. As our machines increasingly read and write without us, as our machines become more and more unreadable, so that seeing no longer guarantees knowing (if it ever did), we the so-called users are offered more to see, more to read. The computer—that most nonvisual and nontransparent device—has paradoxically fostered "visual culture" and "transparency."

Software—or, to be precise, the curious separation of software from hardware—drives this compensatory gesture. Software perpetuates certain notions of seeing as knowing, of reading and readability that were supposed to have faded with the waning of indexicality. It does so by mimicking both ideology *and* ideology critique, by conflating executable with execution, program with process, order with action.[3] Software, through programming languages that stem from a gendered system of command and control, disciplines its programmers and users, creating an invisible system of visibility. The knowledge software offers is as obfuscatory as it is revealing. Thus, if as Lev Manovich recommends in *Language of New Media,* new media studies needs to engage software, it must not merely adopt the language of software but must critically examine the limitations of "transcoding" and software's new status as common sense.[4]

Materializing the Immaterial

Software is, or should be, a notoriously difficult concept. The current commonsense computer science definition of software is a "set of instructions that direct a computer to do a specific task." As a set of instructions, its material status is unstable; indeed, the more you dissect software, the more it falls away. Historian Paul Ceruzzi likens it to an onion, "with many distinct layers of software over a hardware core."[5] This onionlike structure, however, is itself a programming effect: one codes by using another software program; software and hardware (like genes and DNA) cannot be physically separated. Computer scientist Manfred Broy describes software as "almost intangible, generally invisible, complex, vast and difficult to comprehend." Because software is "complex, error-prone, and difficult to visualize," Broy argues, many of its "pioneers" have sought to make "software easier to visualize and understand, to represent the phenomena encountered in software development in models that make the often implicit and intangible software engineering tasks explicit."[6] Friedrich Kittler has more forcefully argued, "there is no software" since everything reduces to voltage differences as signifiers.[7]

In the 1940s software did not exist: there literally was no software.[8] "Programming" comprised the human task of making connections, setting switches, and inputting values ("direct programming"), as well as the human and machine task of coordinating the various parts of the computer. In 1946 the master programmer for the ENIAC (the first general-purpose electronic digital computer to be designed, built, and successfully used) controlled the sequence of actions needed to solve a problem numerically.[9] The ENIAC was initially rewired for each problem so that, essentially, a new ENIAC was created each time it was used. Its conversion to a stored-program computer in 1947 (in part due to a suggestion by John von Neumann) meant that programs could be coded by setting switches, which corresponded to sixty stored instructions, rather than by plugging cables. This change, seen as a way to open up programming to simple scientists, dramatically decreased the time necessary for programming while increasing the time necessary for computation. Today these changeable settings would be called software because, with symbolic programming languages, these physical settings (which for instance enabled a value X to be moved from memory location Y into the accumulator) became translated into a string of numbers read into the computer's memory. Today the clerical "operators" who planned the wiring for and wired the ENIAC (Kathleen McNulty, Frances Bilas, Betty Jean Jennings, Elizabeth Snyder, Ruth Lichterman, and Marlyn Wescoff) are reclaimed as some of the earliest programmers.

Symbolic programming languages and therefore software, as Paul Ceruzzi and Wolfgang Hagen have argued, were not foreseen. The emergence of symbolic-language programming depended on the realization that the computer could store numerical instructions as easily as it could data and that the computer itself could be used to translate between symbolic and numeric notations. The programmers of the EDSAC, an early (1949) computer in Cambridge, England, were the first to use the computer to translate between more humanly readable assembly code (for instance, "A100" for "add the contents of location 100 to the add register") and what has since been called machine language (rather than a logical code). Storage was key to the emergence of programming languages, but, as the case of John von Neumann reveals, storage was not enough: von Neumann, whose name has become the descriptor for all modern stored-program computers also devised a notation similar to the EDSAC's with Herman Goldstine but assumed that clerks would do the translation.[10] Further assembly language is not a higher-level programming language; a computer is not automatically a media machine. According to Hagen, "for decades, the arche-structure of the von Neumann machine did not reveal that this machine would be more than a new calculator, more than a mighty tool for mental labor, namely a new communications medium." The move from calculator to communications medium, Hagen argues, itself stemmed from a "communications imperative" that

> grew out of the cold war, out of the economy, out of the organization of labor, perhaps out of the primitive numeric seduction the machines exerted, out of the numbers game, out of a game with digits, placeholders, *fort/da* mechanisms, and the whole quasi-linguistic *quid pro quo* of the interior structure of all these sources.[11]

Automatic Programming

Automatic programming, what we call programming today, arose from a desire to reuse code and to recruit the computer into its own operation—that is, to transform singular instructions into a language a computer could write. As Mildred Koss, an early UNIVAC programmer, explains:

> Writing machine code involved several tedious steps—breaking down a process into discrete instructions, assigning specific memory locations to all the commands, and managing the I/O

buffers. After following these steps to implement mathematical routines, a sub-routine library, and sorting programs, our task was to look at the larger programming process. We needed to understand how we might reuse tested code and have the machine help in programming. As we programmed, we examined the process and tried to think of ways to abstract these steps to incorporate them into higher-level language. This led to the development of interpreters, assemblers, compilers, and generators—programs designed to operate on or produce other programs, that is, automatic programming.[12]

Automatic programming is an abstraction that allows the production of computer-enabled human-readable code—key to the commodification and materialization of software and to the emergence of higher-level programming languages. This automation of programming—in particular, programming languages—makes programming problem- rather than numerically oriented. Higher-level programming languages, unlike assembly language, explode one's instructions and enable one to forget the machine. They enable one to run a program on more than one machine—a property now assumed to be a "natural" property of software. Direct programming" led to a unique configuration of cables; early machine language could be iterable but only on the same machine—assuming, of course, no engineering faults or failures. In order to emerge as a language or a text, software and the "languages" on which it relies had to become iterable. With programming languages, the product of programming would no longer be a running machine but rather this thing called software—something theoretically (if not practically) iterable, repeatable, reusable, no matter who wrote it or what machine it was destined for. Programming languages inscribe the absence of both the programmer and the machine in its so-called writing.[13] Programming languages enabled the separation of instruction from machine, of imperative from action.

According to received wisdom, these first attempts to automate programming were inefficient and resisted by "real" programmers. John Backus, developer of FORTRAN, claims that early machine language programmers were engaged in a "black art"; they had a "chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals."[14] Koss similarly argues, "without these higher-level languages and processes . . . , which democratized problem solving with the computer, I believe programming would have remained in the hands of a relatively small number of technically oriented software writers using machine code, who would have been essentially the high priests of computing."[15]

The resistance to automatic programming also seems to have stemmed from corporate and academic customers, for whom programmers were orders of magnitude cheaper per hour than computers. Jean Sammett, an early female programmer, relates in her influential *Programming Languages: History and Fundamentals,*

> customers raised many objections, foremost among them was that the compiler probably could not turn out object code as good as their best programmers. A significant selling campaign to push the advantages of such systems was underway at that time, with the spearhead being carried for the numerical scientific languages (i.e., FORTRAN) by IBM and for "English-language-like" business data processing languages by Remington Rand (and Dr. Grace Hopper in particular).[16]

This selling campaign not only pushed higher-level languages (by devaluing humanly produced programs), it also pushed new hardware: to run these programs, one needed more powerful machines. The government's insistence on standardization, most evident in the development and dissemination of COBOL, also greatly influenced the acceptance of higher-level languages, which again were theoretically, if not always practically, machine independent or iterable. The hardware-upgrade cycle was normalized in the name of saving programming time.

The "selling campaign" led to what many have heralded as the democratization of programming. In Sammet's view, this was a partial revolution,

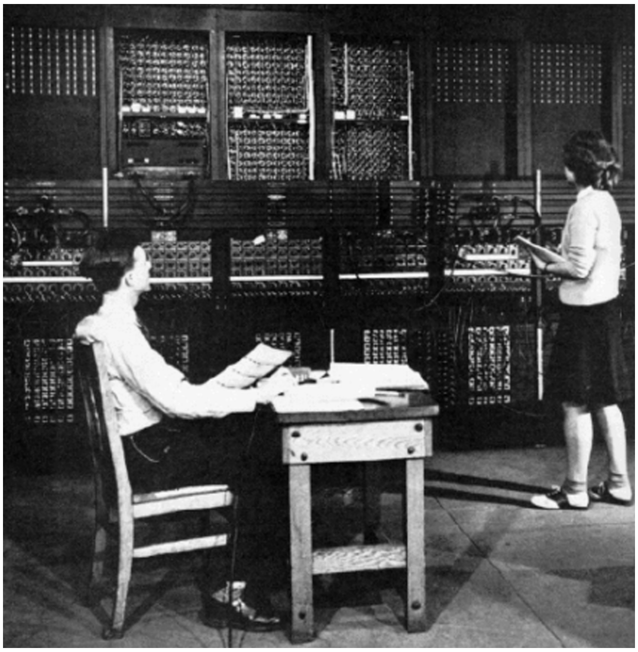
> in the way in which computer installations were run because it became not only possible, but quite practical to have engineers, scientists, and other people actually programming their own problems without the intermediary of a professional programmer. Thus the conflict of the open versus closed shop became a very heated one, often centering around the use of FORTRAN as the key illustration for both sides. This should not be interpreted as saying that all people with scientific numerical problems to solve immediately sat down to learn FORTRAN; this is clearly not true but such a significant number of them did that it has had a major impact on the entire computer industry. One of the subsidiary side effects of FORTRAN was the introduction of FORTRAN Monitor System [IB60]. This made the computer installation much more efficient by requiring less operator intervention for the running of the vast number of FORTRAN (as well as machine language) programs.[17]

This "opening" of computing, which gives the term *open* in "open source" a different resonance, would mean the potential spread of computing to those with scientific numerical problems to solve and the displacement of human operators by operating systems. But scientists have always been involved with computing, even though computing has not always been considered to be a worthy scientific pursuit and, as mentioned previously, the introduction of dials rather than wires was supposed to empower simple scientists. The history of computing is littered with moments of "computer liberation."[18]

This narrative of the "opening" of computing through higher-level languages assumes that programmers naturally enjoyed tedious and repetitive numerical tasks and developing singular solutions for their clients. The "mastery" of computing can easily be understood as "suffering." Harry Reed, an early ENIAC programmer, argues:

> The whole idea of computing with the ENIAC was a sort of *hair-shirt* kind of thing. Programming for the computer, whatever it was supposed to be, was a redemptive experience—one was supposed to *suffer* to do it. And it wasn't until the 1970s that we finally were able to convince people that they were not going to have programmers continually writing little programs for them. I actually had to take my Division and sit everybody down who hadn't take a course in FORTRAN, because, by God, they were going to write their own programs now. We weren't going to get computer specialists to write simple little programs that they should have been writing.[19]

The narrative of the democratization of programming reveals the tension at the heart of programming and control systems: are they control systems or servomechanisms (Norbert Wiener's initial name for them)? Is programming a clerical activity or an act of mastery? Given that the machine takes care of "programming proper"—the sequence of events during execution—is programming programming at all? What is compacted in the *linguistic* move from "operator" to "programmer"? The notion of a "priesthood" of programmers erases this tension, making programming always already the object of jealous guardianship, and erasing programming's clerical underpinnings. Programming in the 1950s does seem to have been fun and fairly gender balanced, in part because it was so new and in part because it was not as lucrative as hardware design or sales: the profession was gender neutral in hiring if not pay because it was not yet a profession.[20] The "ENIAC girls" were first hired as subprofessionals, and some had to acquire more qualifications in order to retain their positions. As many female programmers quit to have children or get married, men took their increasingly lucrative jobs. Programming's clerical and arguably feminine underpinnings—both in terms of personnel and command structure—was buried as programming sought to become an



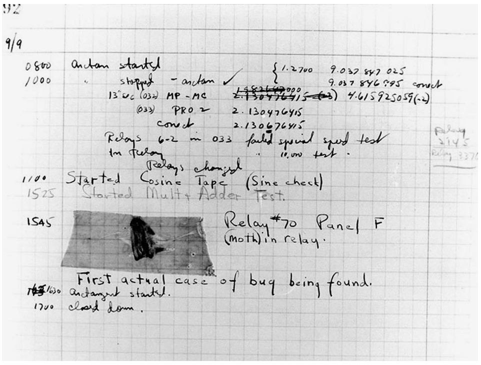
engineering and academic field in its own right. Such erasure is key to the professionalization of programming—a compensatory mastery built on hiding the machine. Democratization did not displace professional programmers but rather buttressed their position as professionals by paradoxically decreasing their real power over their machines and by generalizing the engineering concept of information.

Yes, Sir

The assumption that programmers naturally enjoy tedious tasks points to programming and computing's gendered and human history. During World War II almost all computers were young women with some background in mathematics. Not only were women available for work then, they were also considered to be better, more conscientious computers, presumably because they were better at repetitious, clerical tasks. Programmers were former computers because they were best suited to prepare their successors: they thought like computers.

One could say that programming became programming and software became software when commands shifted from commanding a "girl" to commanding a machine. The image above reveals the dream of "programming proper"—a man sitting at a desk giving commands to a female "operator." Software languages are based on a series of imperatives that stem from World War II command and control structure. The automation of command and control, which Paul Edwards has identified as a perversion of military traditions of "personal leadership, decentralized battlefield command, and experience-based authority,"[21] arguably started with World War II mechanical computation. This is most starkly exemplified by the relationship between the Wrens, volunteer members of the Women's Royal Naval Service, and their commanding officers at Bletchley Park. The Wrens, also called slaves by Turing (a term now embedded within computer systems), were clerks responsible for the mechanical operation of the cryptanalysis machines (the Bombe and then the Colossus), although at least one of the clerks, Joan Clarke, became an analyst. Revealingly, I. J. Good, a male analyst, describes the Colossus as enabling a man-machine synergy duplicated by modern machines only in the late 1970s: "the analyst would sit at the typewriter output and call out instructions to a Wren to make changes in the programs. Some of the other uses were eventually reduced to decision trees and were handed over to the machine operators (Wrens)."[22] This man-machine synergy,

ENIAC programmers, late 1940s. U.S. Military Photo, Redstone Arsenal Archives, Huntsville, Alabama.



92

9/9

0800 arctan started

1000 " stopped - arctan ✓ {1.2700 9.037 847 025 / 9.037 846 995 correct}

13" sec (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(23)~~) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

~~143~~ 1630 arctangent started.

1700 closed down.

Relay 2145

Relay 3370

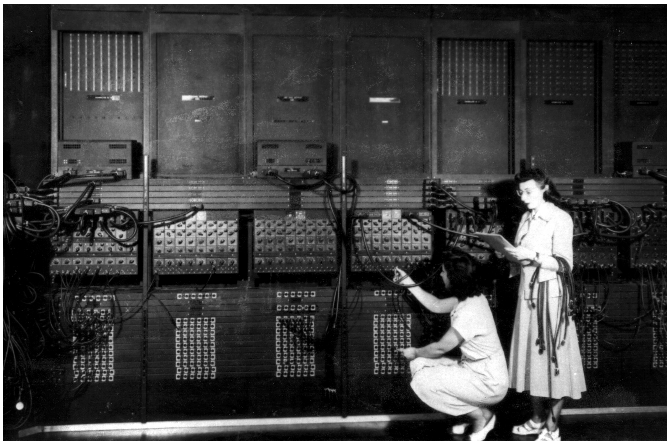
or interactive real-time (rather than batch) processing, treated Wrens and machines indistinguishably, while simultaneously relying on the Wrens' ability to respond to the mathematician's orders.

The story of the initial meeting between Grace Murray Hopper (one of the first and most important programmers) and Howard Aiken also buttresses this narrative. Hopper, a Ph.D. in mathematics from Yale and a former mathematics professor at Vassar, was assigned by the U.S. Navy to program the Mark I, an electromechanical digital computer that made a sound like a roomful of knitting needles. According to Hopper, Aiken showed her

> a large object with three stripes . . . waved his hand and said: "That's a computing machine." I said, "Yes, Sir." What else could I say? He said he would like to have me compute the coefficients of the arc tangent series, for Thursday. Again, what could I say? "Yes, Sir." I didn't know what on earth was happening, but that was my meeting with Howard Hathaway Aiken.[23]

Computation depends on "yes, sir" in response to short declarative sentences and imperatives that are in essence commands. Contrary to Neal Stephenson, in the beginning was the command rather than the command line. The command line is a mere operating system (OS) simulation. Commands have enabled the slippage between programming and action that makes software such a compelling yet logically "trivial" communications system. I.J. Good's and Hopper's recollections also reveal the routinization at the core of programming: the analyst at Bletchley Park was soon replaced by decision trees. Hopper the programmer became an advocate of automatic programming. Thus routinization or automation lies at the core of a profession that likes to believe it has successfully automated every profession but its own.[24]

But to view women and mechanical computation as interchangeable is to revise history. According to Sadie Plant, programming is essentially feminine—not simply because women, from Ada Lovelace to Hopper, were the first programmers, but because of the historical and theoretical ties between programming and what Freud called the quintessentially feminine invention of weaving, between female sexuality as mimicry and the mimicry grounding Turing's vision of computers as universal machines. (In addition, both software and feminine sexuality reveal the power that something which cannot be seen can have.)[25] Women, Plant argues,



have not merely had a minor part to play in the emergence of digital machines. . . . Theirs is not a subsidiary role which needs to be rescued for posterity, a small supplement whose inclusion would set the existing records straight . . . . Hardware, software, wetware—before their beginnings and beyond their ends, women have been the simulators, assemblers, and programmers of the digital machines.[26]

The photograph on page 33 is not representative—the female programmers of the ENIAC worked together in pairs, and no machine could have accomplished what Hopper did—at least not then. (And again, Hopper would be key to enabling computers to do so: the closure of the distance between Hopper and computers would be key to automatic command and control). The difficulty faced by programmers was simple: computers weren't computers. The transition from commanding a girl to commanding an automaton was difficult because automatons deciphered rather than interpreted or presumed, and they did not learn from experience. As Martin Campbell-Kelly and William Aspray put it, "the fundamental difficulty of writing software was that, until computers arrived, human beings had never before had to prepare detailed instructions for an automaton—a machine that obeyed unerringly the commands given to it, and for which every possible outcome had to be anticipated by the programmer."[27] Campbell-Kelly and Aspray's assessment overestimates the reliability of the machines, especially the early ones. As the early ENIAC programmers relate, part of debugging was figuring out which errors stemmed from programming and which from faulty vacuum tubes, accidental rewiring, or faults in machine architecture—a task made easier by neon bulbs attached to various counters.

Unlike machines, women programmers did not simply follow instructions. Hopper was described as "a woman of strong personality, a powerful persuader and leader. She showed some of her Navy training in her commanding speech."[28] Also, programming, as Koss explains, was not just implementing instructions but "designing a strategy and preparing instructions to make the computer do what you wanted it to do to solve a problem."[29] Women programmers played an important role in converting the ENIAC into a stored-program computer and in determining the trade-off between storing values and instructions. Betty Snyder Holberton, described by Hopper as the best programmer she had known, not only debugged the ballistics-trajectory program in a dream (the first program to be run on the ENIAC, albeit too late to be of use for WWII), she also developed an influential sort

Opposite, top: Captain Grace M. Hopper, 1976. U.S. Navy Photo (number NH 96945), Naval Historical Center, Washington, D.C.

Opposite, bottom: First Computer "Bug," 1945. U.S. Navy Photo (number NH 96566-KN), Naval Historical Center, Washington, D.C.

Above: Two women wiring the right side of the ENIAC with a new program, late 1940s. U.S. Army Photo, Army Research Laboratory Technical Library, Aberdeen Proving Ground, Aberdeen, Maryland.

algorithm for the UNIVAC.[30]

The extent of the repression of these women in standard histories of computing can be measured by Paul Edwards's implicit, a priori gendering of the computer "work force" in an otherwise insightful analysis of masculinity and programming. He writes:

> computer scientists enjoy a mystique of hard mastery comparable to the cult of physicists in the postwar years. Computers provide them with unblinking precision, calculative power, and the ability to synthesize massive amounts of data. . . .
>
> There is nothing inherently masculine about computer technology. Otherwise women could not have had such quick success in *joining the computer work force.* Gender values largely float free of the machines themselves and are expressed and enforced by power relationships between men and women. Computers do not simply embody masculinity; they are culturally constructed as masculine mental objects.[31]

A far cry from the claim of J. Chuan Chu (one of the original ENIAC hardware engineers) that software is the daughter of Frankenstein (hardware being its son), Edwards's assessment erases the overwhelming presence of women in early computing—their work as human computers, programmers, and monitors—while it reduces computer technology to software. As the combination of a human clerk and a human computer, the modern computer encapsulates the power relations between men and women in the 1940s. It sought to displace women: their nimble fingers, their numerical abilities, their discretion, their "disquieting gazes"—a displacement Vannevar Bush viewed as desirable.[32] The transition from human to mechanical computers automated differential power relationships.

Recognizing these women as programmers—as not merely following but also putting together instructions—is important but not enough, for it keeps in place the narrative of programming as "masterful." What is the significance of following and implementing instructions? Perhaps the "automation" of control and command is less a perversion of military tradition and more an instantiation of it, one in which responsibility has been handed over to those (now machines) implementing commands. The relationship between masters and slaves is always ambiguous. This handing over of power has been hidden by programming languages that obscure the machine and highlight programming (rather than execution) as the source of action. The closing of the distinction between programming and execution, evidenced in the ambiguity of the object of the verb "to program," was facilitated by the disciplining and professionalization of pro-

The First Four, 1950s. U.S. Army Photo (number 163-12-62).

grammers through "structured programming."

Hiding the Machine

During the much discussed "software crisis" of the late 1960s, which stemmed from such spectacular debacles as IBM's OS/360, many (especially European programmers, such as Friedrich [Fritz] Bauer and Peter Naur) viewed "software engineering," or structured programming, as a way to move programming from a craft to a standardized industrial practice, and as a way to create disciplined programmers who dealt with abstractions rather than numerical processes.[33] As Michael Mahoney has argued, structured programming emerged as a "means both of quality control and of disciplining programmers, methods of cost accounting and estimation, methods of verification and validation, techniques of quality assurance."[34]

"Structured programming" (also generally known as "good programming") hides, and thus secures, the machine. Not surprisingly, having little to no contact with the actual machine enhances one's ability to think abstractly rather than numerically. Edsger Dijkstra, whose famous condemnation of "go to" statements has encapsulated to many the fundamental tenets of structure programming, believes that he was able to "pioneer" structured programming precisely because he began his programming career by coding for machines that did not yet exist.[35] In "Go To Statement Considered Harmful," Dijkstra argues, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce." This is because go to statements go against the fundamental tenet of good programming—the necessity to "shorten the conceptual gap between static program and dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible." More specifically, if a program is halted, go tos make it difficult to find a place in the programming that corresponds to the halted process—it makes it "terribly hard to find a meaningful set of coordinates in which to describe the process progress."[36] That is, go tos make difficult the conflation of instruction with command, which grounds "programming."[37]

Structured programming languages "save" programmers from themselves by providing good security, where security means secure from the programmer.[38] In the name of security, structured programming, which emphasizes programming as a question of flow, is itself giving way to data abstraction, which views programming as a question of interrelated objects and hides far more than the machine. Data abstraction depends on information hiding, on the nonreflection of changeable facts in software. As John V. Guttag, a "pioneer" in data abstraction explains, data abstraction is all about forgetting.[39] Rather than "polluting" a program by enabling invisible lines of contact between supposedly independent modules, data abstraction presents a clean or "beautiful" interface by confining specificities, and by reducing the knowledge and power of the programmer. Knowledge, Guttag insists, is dangerous: "'Drink deep, or taste not the Pierian Spring,' is not necessarily good advice. Knowing too much is no better, and often worse, than knowing too little. People cannot assimilate very much information. Any programming method or approach that assumes that people will understand a lot is highly risky."[40]

Thus abstraction both empowers the programmer and insists on his/her ignorance. Because abstraction exists "in the mind of the programmer," abstraction gives programmers new creative abilities. Computer scientist David Eck argues, "every programming language defines a virtual machine, for which it is the machine language. Designers of programming languages are creating computing machines as surely as the engineer who works in silicon and copper, but without the limitations imposed by materials and manufacturing technology."[41] However, this abstraction—this drawing away from the specificities of the machine—gives over, in its separation of machine into software and hardware, the act of programming to the machine itself. Koss scoffed at the early notion of computers as brains because "they couldn't think in the way a human thinks, but had to be given a set of step-by-step machine instructions to be executed before they could provide answers to a specific problem"—at that time software was not considered to be an independent object.[42] The current status of software as a commodity, despite the fact that its instructions are immaterial and nonrivalrous indicates the triumph of the software industry, an industry that first struggled not only financially but also conceptually to define its product. The rise of software depends both on historical moves, such as IBM's unbundling of its services from its products, and on abstractions enabled by higher-level languages. Guttag's insistence on the unreliability and incapability of human beings to understand underscores the costs of such an abstraction. Abstraction is the computer's game, as is programming in the strictest sense of the word.

Importantly, programmers are *users:* they create programs using editors, which are themselves software programs. The distinction between programmers and users is gradually eroding, not only because users are becoming programmers (in a real sense programmers no longer program a computer; they code), but also because, with high-level languages, programmers are becoming more like simple users. The difference between users and programmers is an effect of software.

Causal Pleasure

The gradual demotion of programmers has been offset by the power and pleasure of programming. As Edwards argues, "programming can produce strong sensations of power and control" because the computer produces an internally consistent if externally incomplete microworld,

> a simulated world, entirely within the machine itself, that does not depend on instrumental effectiveness. That is, where most tools produce effects on a wider world of which they are only a part, the computer contains its own worlds in miniature. . . . In the microworld, as in children's make-believe, the power of the programmer is absolute.[43]

This pleasure is itself an effect of programming languages, which offer the lure of visibility, readability, cause and effect. Consider this ubiquitous "hello world" program written in C++ ("hello world" is usually the first program a person will write):

```
// this program spits out "hello world"
#include <iostream.h>
int main ()
{
    cout << "Hello World!";
    return 0;
}
```

The first line is a comment line, explaining to the human reader that this program spits out "hello world." The next line directs the compiler's pre-processor to include iostream.h, a standard file to deal with input and output. The third line, "int main ()", begins the main function of the program; "cout<<"Hello World!";" prints "Hello World" to the screen ("cout" is defined in iostream.h); "return 0" terminates the main function and causes the program to return a 0 if it has run correctly. Although not immediately comprehensible to someone not versed in C++, this program nonetheless seems to make some sense. It comprises a series of imperatives and declaratives that the computer presumably understands and obeys. When it runs, it follows one's commands and displays "Hello World." Importantly, this message, which affirms the programmer's agency, also calls it into question: who or what, after all, is saying "hello world?" To enjoy this absolute power, the programmer must follow the rules of a programming language. Regardless, seeing his or her code produce visible and largely predictable results creates pleasure.[44] One's code causes an action to happen: cause and effect is clear, even if the end result is never entirely predictable. This absolute power enabled through the agency of a program reveals the contradictory status of agency, namely the fact that agency refers both to one's ability to act and the ability of someone else to act on one's behalf.

In the microworld of computer simulation, however, it is not only the programmer whose power is enhanced or absolute, for interactive simulations—key to the concept of computers as transparent—enhance the power of the user (again, these terms are not absolute but rather depend on the software one is using and on the productive power of one's actions). Interactivity, intimately linked, as Edwards has argued, to artificial intelligence, stemmed initially from an admission of human fallibility, and of the limitations of procedural programming languages.[45] By the 1960s, the naïveté behind von Neumann's assertion that "anything that can be exhaustively and unambiguously described, anything that can be completely and unambiguously put into words, is ipso facto realizable by a suitable finite neural network" was becoming increasingly apparent.[46] Since exhaustive and unambiguous description was difficult, if not impossible, working "interactively" with a computer to solve problems was key. Interactivity later became conflated with user freedom, and control with GUI (graphical user) and WYSIWYG (What You See Is What You Get) interfaces, which were seen as supplements to language-based commands. Unlike command-line interfaces, GUIs enabled "direct manipulation," in which mastery was intimately linked to simulated visibility. According to Ben Schneiderman:

> Certain interactive systems generate glowing enthusiasm among users—in marked contrast with the more common reaction of grudging acceptance or outright hostility. The enthusiastic users' reports are filled with positive feelings regarding:
>
> - mastery of the system
> - competence in the performance of their task
> - ease in learning the system originally and in assimilating advanced features
> - confidence in their capacity to retain mastery over time
> - enjoyment in using the system
> - eagerness to show it off to novices, and
> - desire to explore more powerful aspects of the system
>
> These feelings are not, of course, universal, but the amalgam does convey an image of the truly pleased user. . . . The central ideas seemed to be visibility of the object of interest; rapid, reversible, incremental actions, and, replacement of complex command language syntax by direct manipulation of the object of interest—hence the term "direct manipulation."[47]

As Brenda Laurel has argued in her comparison of computer interfaces and theater, direct manipulation (which is anything but direct) must be complemented by direct engagement in order to be successful. Direct engagement, Laurel argues

> shifts the focus from the representation of manipulable objects to the ideal of enabling people to engage directly in the activity of choice, whether it be manipulating symbolic tools in the performance of some instrumental tasks or wandering around the imaginary world of a computer game. Direct engagement emphasizes emotional as well as cognitive values. It conceives of human-computer activity as a *designed experience.*[48]

Laurel's emphasis on action underscores the crucial difference between the representation of tools and the tools themselves: she argues that people realize when they double click on a folder that it is not really a folder, and making a folder more "life-like" is not helpful. What is helpful, Laurel contends, is clear causality: events must happen in such a way that the user can accept them as probable and thus narrow probability into certainty. Causality, she claims, ensures universality, ensures that the users will willingly suspend their disbelief. For users as for paranoid schizophrenics (my observation, not Laurel's), everything has meaning: there can be no coincidences, only causal pleasure.

Causal pleasure is not simply a representation of user actions in a causally plausible manner; it is also a "user amplification." Manovich explains "user amplification" in terms of Super Mario:

> when you tell Mario to step to the left by moving a joystick, this initiates a small delightful narrative: Mario comes across a hill; he starts climbing the hill; the hill turns out to be too steep; Mario slides back onto the ground; Mario gets up, all shaking. None of these actions required anything from us; all we had to do is just to move the joystick once. The computer program amplifies our single action, expanding it into a narrative sequence.[49]

This user amplification mimics the "instruction explosion" driving higher-level programming languages (one line of high-level code corresponds to more than one line of machine code); user amplification is not only the product of gaming software and software art but is central to the power of programming.

This dual amplification arguably drives the romanticization of programming and, more recently, the emergence of software art, or Generation Flash. According to Manovich, Generation Flash is a new group of artists ("new romantics") who create original code rather than participate in the endless cycle of postmodern citation. As programmers Generation Flash artists produce

> the new modernism of data visualizations, vector nets, pixel-thin grids and arrows: Bauhaus design in the service of information design. Instead [of] the Baroque assault of commercial media, Flash

> generation serves us the modernist aesthetics and rationality of software. Information design is used as a tool to make sense of reality while programming becomes a tool of empowerment.[50]

To make sense of reality, these artists and designers employ user amplification, for their modernist aesthetics and software-based rationality amplify and simplify cause and effect. In line with software more generally, they unveil—they make things visible. This unveiling depends on a certain "smartness" on the part of the user. Describing Futurefarmer's project theyrule.net, which offers users a way to map the relationships between board members of the most powerful corporations, Manovich states:

> Instead of presenting a packaged political message, it gives us data and the tools to analyze it. It knows that we are intelligent enough to draw the right conclusion. This is the new rhetoric of interactivity: we get convinced not by listening/watching a prepared message but by actively working with the data: reorganizing it, uncovering the connections, becoming aware of correlations.[51]

According to Manovich, this new rhetoric of interactivity is further explored in UTOPIA:

> The cosmogony of this world reflects our new understanding of our own planet—post Cold War, Internet, ecology, Gaia, and globalisation. Notice the thin barely visible lines that connect the actors and the blocks. (This is the same device used in theyrule.net.) In the universe of UTOPIA, everything is interconnected, and each action of an individual actor affects the system as a whole. Intellectually, we know that this is how our Earth functions ecologically and economically—but UTOPIA represents this on a scale we can grasp perceptually.[52]

UTOPIA enables what Fredric Jameson has called a "cognitive map": "a situational representation on the part of the individual subject to that vaster and properly unrepresentable totality which is the ensemble of society's structures as a whole."[53] If cognitive mapping is both difficult and necessary now because of invisible networks of capital, these artists produce a cognitive map by exploiting the invisibility of information. The functioning of software art, as Manovich argues, parallels Marxist ideology critique. The veil of ideology is torn asunder by grasping the relations between the action of individual actors and the system as a whole. Software enables this critique by representing it at a scale—in a microworld—that we can make sense of. This unveiling depends on our own actions, on us manipulating objects in order to see, on us thinking like object-oriented programmers. Rather than lack cognitive maps, we produce them all the time through a medium that simulates ideology critique and, in its nonexistence, ideology as well. It is truly remarkable that software—designed to obfuscate the machine and create a virtual one and based on buried commands—has led to the overwhelming notion of computation as transparent. This notion of transparency has less to do with actual technological operations than with the "microworld" established by computation.

Software as Ideology

As I've argued elsewhere, software is a functional analog to ideology.[54] In a *formal* sense computers understood as comprising software and hardware are ideology machines. They fulfill almost every formal definition of ideology we have, from ideology as false consciousness (as portrayed in *The Matrix*) to Louis Althusser's definition of ideology as "a 'representation' of the imaginary relation of individuals to their real conditions of existence."[55] Software, or perhaps more precisely operating systems, offer us an imaginary relationship to our hardware: they do not represent transistors but rather desktops and recycling bins. Software produces "users." Without OS there would be no access to hardware; without OS no actions, no practices, and thus no user. Each OS, through its advertisements, interpellates a "user": calls it and offers it a name or image with which to identify. So Mac users "think different" and identify with Martin Luther King and Albert Einstein; Linux users are open-source power geeks, drawn to the image of a fat, sated penguin; and Windows users are mainstream, functionalist types perhaps comforted, as Eben Moglen argues, by their regularly crashing computers. Importantly, the "choices" operating systems offer limit the visible and the invisible, the imaginable and the unimaginable. You are not, however, aware of software's constant constriction and interpellation (also known as its "user-friendliness"), unless you find yourself frustrated with its defaults (which are remarkably referred to as *your* preferences) or you use multiple operating systems or competing software packages.

Software also produces users through benign interactions, from reassuring sounds that signify that a file has been saved to folder names such as "my documents," which stress personal computer ownership. Computer programs shamelessly use shifters, pronouns like "my" and "you," that address you, and everyone else, as a subject. Software makes you read, offers you more relationships and ever more visuals. Software provokes readings that go beyond reading letters toward the nonliterary and archaic practices of guessing, interpreting, counting, and repeating. Software is based on a fetishistic logic.[56] Users know very well that their folders and desktops are not really folders and desktops, but they treat them as if they were—by referring to them as folders and as desktops. This logic is, according to Slavoj Zizek, crucial to ideology. Zizek (through Peter Sloterdjik) argues that ideology persists in one's actions rather than in one's beliefs. The illusion of ideology exists not at the level of knowledge but rather at the level of doing: this illusion, maintained through the imaginary "meaning of the law" (causality), screens the fact that authority is without truth—that one obeys the law to the extent that it is incomprehensible. *Is this not computation?* Through the illusion of meaning and causality do we not cover over the fact that we do not and cannot fully understand nor control computation? That computers increasingly design each other and that our use is—to an extent—a supplication, a blind faith? The new rhetoric of "interactivity" obfuscates more than it reveals.

Operating systems also create users more literally, for users are an OS construction. User logins emerged with time-sharing operating systems, like UNIX, which encourages users to believe that the machines they are working on are their own machines (before this, computers mainly used batch processing; before that, one really did run the computer, so there was no need for operating systems—one had human operators). As many historians have argued, the time-sharing operating systems developed in the 1970s spawned the "personal computer."[57]

Software and ideology fit each other perfectly because both try to map the material effects of the immaterial and to posit the immaterial through visible cues. Through this process the immaterial emerges as a commodity, as something in its own right. Thus Broy's description of pioneers as seeking to make software easier to visualize strangely parallels software itself, for what is software if not the very effort of making something explicit, of making something intangible visible, while at the same time rendering the visible (such as the machine) invisible? Although the parallel between software and ideology is compelling, it is important that we not rest here, for reducing ideology to software empties ideology of its critique of power—something absolutely essential to any theory of ideology.[58] The fact that software, with its onionlike structure, acts both as ideology *and* ideology critique—as a concealing and a means of revealing also breaks the analogy between software and ideology. The power of software lies with this dual action and the visible it renders invisible, an effect of programming languages becoming a linguistic task.

Seeing through Transparency

When you draw a rabbit out of a hat, it's because you put it there in the first place.

—Jacques Lacan[59]

This act of revealing drives databases and other structures key to "transparency" or what Baudrillard called the "obscenity" of communication. Although digital imaging certainly plays a role in the notion of computer networks as transparent, it is not the only, nor the key, thing. Consider, for instance, "The Matrix," a multistate program that sifts through databases of public and private information ostensibly to find criminals or terrorists. This program works by integrating

> information from disparate sources, like vehicle registrations, driver's license data, criminal history and real estate records and analyzing it for patterns of activity that could help law enforcement investigations. Promotional materials for the company argued, "When enough seemingly insignificant data is analyzed against billions of data elements, the invisible becomes visible."[60]

Although supporters claim that "the Matrix" simply brings together information already available to law enforcement,

> opponents of the program say the ability of computer networks to combine and sift mountains of data greatly amplifies police surveillance power, putting innocent people at greater risk of being entangled in data dragnets. The problem is compounded, they say, in a world where many aspects of daily life leave online traces.[61]

By March 15, 2004, over two-thirds of the states withdrew their support for "The Matrix," citing budgetary and privacy concerns. "The Matrix" was considered to be a violation of privacy because it made the invisible visible (again, the act of software itself), not because the computer reproduced indexical images. It amplified police power by enabling them to make easy connections. The Total Information Agency, a U.S. government plan to bring together its various electronic databases, was similarly decried and basically killed by the U.S. Congress in 2003.

On a more personal level, computing as enabling connections through rendering the invisible visible drives personal computing interfaces. By typing in Word, letters appear on my screen, representing what is stored invisibly on my computer. My typing and clicking seem to have corresponding actions on the screen. By opening a file, I make it visible. On all levels, then, software seems about making the invisible visible—about *translating* between computer-readable code and human-readable language. Manovich seizes on this translation and makes "transcoding"—the translation of files from one format to another, which he extrapolates to the relationship between cultural and computer layers—his fifth and last principle of new media in *The Language of New Media*. Manovich argues that in order to understand new media we need to engage both layers, for although the surface layer may seem like every other media, the hidden layer, computation, is where the true difference between new and old media—programmability—lies. He thus argues that we must move from media studies to software studies, and the principle of transcoding is one way to start to think about software studies.[62]

The problem with Manovich's notion of transcoding is that it focuses on static data and treats computation as a mere translation. Programmability does not only mean that images are manipulable in new ways but also that one's computer constantly acts in ways beyond one's control. To see software as merely "transcoding" erases the computation necessary for computers to run. The computer's duplicitous reading does not merely translate or transcode code into text/image/sound or vice versa; its reading—which conflates reading and writing (for a computer, to read is to write elsewhere)—also partakes in other invisible readings. For example, when Microsoft's Media Player plays a CD, it sends the Microsoft Corporation information about that CD. When it plays a Real Media file, such as a CNN video clip, it sends CNN its "unique identifier." You can choose to work off-line when playing a CD and request that your media player not transmit its "unique identifier" when online, but these choices require two changes to the default settings. By installing the Media Player, you also agreed to allow Microsoft to "provide security related updates to the OS Components that will be automatically downloaded onto your computer. These security related updates may disable your ability to copy and/or play Secure Content and use other software on your computer."[63] Basically, Microsoft can change components of your operating system without notice or your explicit consent. Thus to create a more "secure" computer, where secure means secure *from* the user, Microsoft can disable pirated files and applications and/or report their presence to its main database.[64] Of course Microsoft's advertisements do not emphasize the Media Player's tracking mechanisms but rather sell it as empowering and user friendly. Now you can listen to both your CD and Internet-based radio stations with one click of a mouse: it is just like your boom box, but better. Now you can automatically receive software updates and optimize your connection to remote sites.

To be clear: this article is not a call to a return to an age when one could see what one does. Those days are long gone. As Kittler argues, at a fundamental level we no longer write—through our use of word processors we have given computers that task.[65] Neither is it an indictment of software or programming (I too am swayed by and enamored of the causal pleasure of software). It is, however, an argument against commonsense notions of software precisely because of their status as common sense (and in this sense they fulfill the Gramscian notion of ideology as hegemonic common sense); because of the histories and gazes they erase; and because of the future they point toward. Software has become a commonsense shorthand for culture and hardware a shorthand for nature. (In the current debate over stem cell research, stem cells have been called “hardware.” Historically software also facilitated the separation of pattern from matter, necessary for the separation of genes from DNA.[66]) In our so-called postideological society, software sustains and depoliticizes notions of ideology and ideology critique. People may deny ideology, but they don’t deny software—and they attribute to software, metaphorically, greater powers than have been attributed to ideology. Our interactions with software have disciplined us, created certain expectations about cause and effect, offered us pleasure and power that we believe should be transferable elsewhere. The notion of software has crept into our critical vocabulary in mostly uninterrogated ways.[67] By interrogating software and the visual knowledge it perpetuates, we can move beyond the so-called crisis in indexicality toward understanding the new ways in which visual knowledge is being transformed and perpetuated, not simply displaced or rendered obsolete.

Notes

1. As quoted by John Schwartz, "Privacy Fears Erode Support for a Network to Fight Crime," *The New York Times,* 15 March 2004, C1. Seisint is a corporation developing "the Matrix," a computer system discussed later in this article.

2. Jean Baudrillard, *The Ecstasy of Communication,* trans. Bernard Schutze and Caroline Schutze (Brooklyn: Semiotext(e), 1988), 21–22; emphasis in original.

3. What, for instance, is Microsoft Word? Is it the encrypted executable, residing on a CD or on one's hard drive (or even its source code), or its execution? The two, importantly, are not the same even though they are both called Word: not only are they in different locations, one is a program while the other is a process. Structured programming, as discussed later, has been key to the conflation of program with process.

4. See Lev Manovich, *The Language of New Media* (Cambridge: MIT Press, 2001), 48.

5. Paul Ceruzzi, *A History of Modern Computing*, 2nd ed. (Cambridge: MIT Press, 2003), 80.

6. Manfred Broy, "Software Engineering—From Auxiliary to Key Technology," in *Software Pioneers: Contributions to Software Engineering,* ed. Manfred Broy and Ernst Denert (Berlin: Springer, 2002), 11–12.

7. Friedrich Kittler, "There Is No Software," ctheory.net, 18 October 1995, http://www.ctheory.net/text_file.asp?pick=74.

8. In the 1950s the term *software* was used infrequently and referred to anything that complemented hardware, such as the plug configuration of cables. In the 1960s *software* became more narrowly understood as what we would now call systems software and more broadly as "any combination of tools, applications, and services purchased from an outside vender." Thomas Haigh, "Application Software in the 1960s as Concept, Product, and Service," *IEEE Annals of the History of Computing* 24, no. 1 (January–March 2002): 6. For more on this see Wolfgang Hagen's "The Style of Source Codes," trans. Peter Krapp in *New Media, Old Media: A History and Theory Reader*, ed. Wendy Hui Kyong Chun and Thomas Keenan (New York: Routledge, 2005).

9. See Herman Goldstine and Adele Goldstine, "The Electronic Numerical Integrator and Computer (ENIAC)" *IEEE Annals of the History of Computing* 18 no. 1 (Spring 1996): 10–16. The two activities have been divided by Arthur Burks into "numerical programming" (implemented by the operators) and "programming proper" (designed by engineers and implemented by the master programming unit). Arthur Burks, "From ENIAC to the Stored-Program Computer: Two Revolutions in Computers," in *A History of Computing in the Twentieth Century,* ed. N. Metropolis et al., 311–344 (New York: Academic Press, Inc., and Harcourt Brace Jovanovich, 1980). Numerical programming dealt with the specifics of the problem and the arithmetic units and was similar to microprogramming; programming proper dealt with the control units and the sequence of operations.

10. See Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 184.

11. See Wolfgang Hagen, "The Style of Source Codes," in *New Media, Old Media,* ed. Wendy Hui Kyong Chun and Thomas Keenan (New York: Routledge, 2005).

12. Adele Mildred Koss, "Programming on the Univac 1: A Woman's Account," *IEEE Annals of the History of Computing* 25, no. 1 (January–March 2003): 56.

13. See Jacques Derrida, "Signature Event Context," in Jacques Derrida, *Limited Inc.,* trans. Samuel Weber and Jeffrey Mehlman, 1–23 (Evanston: Northwestern University Press, 1988).

14. John Backus, "Programming in America in the 1950s—Some Personal Impressions," *A History of Computing in the Twentieth Century,* ed. Metropolis et

al., 127.

15. Koss, 58.

16. Jean Sammett, *Programming Languages: History and Fundamentals* (Englewood Cliffs: Prentice-Hall, 1969), 144.

17. Sammett, 148.

18. See for instance Theodor Nelson, *Computer Lib* (Richmond, WA: Tempus Books of Microsoft Press, 1987).

19. Harry Reed, "My Life with the ENIAC: A Worm's Eye View," in *Fifty Years of Army Computing from ENIAC to MSRC*, Army Research Laboratory, 2000, 158; accessible online at: http://ftp.arl.mil/~mike/comphist/harry_reed.pdf.

20. See "Anecdotes: How Did You First Get into Computing?" *IEEE Annals of the History of Computing* 25, no. 4 (October–December 2003): 48–59.

21. Paul N. Edwards, *The Closed World: Computers and the Politics of Discourse in Cold War America* (Cambridge: MIT Press, 1996), 71.

22. I.J. Good. "Pioneering Work on Computers at Bletchley," in *A History of Computing in the Twentieth Century,* ed. Metropolis et al., 31–46.

23. Quoted in Ceruzzi, 82.

24. See Michael S. Mahoney, "Finding a History for Software Engineering," *IEEE Annals of the History of Computing* 27, no. 1 (January–March 2004): 8–19.

25. As argued earlier, although software produces visible effects, software itself cannot be seen. Luce Irigaray, in *This Sex Which Is Not One* (trans. Catherine Porter, Ithaca, NY: Cornell UP, 1985), has similarly argued that feminine sexuality is non-visual, "her sexual organ represents *the horror of nothing to see";* emphasis in original, 26.

26. Sadie Plant, *Zeros + Ones: Digital Women + The New Technoculture* (New York: Doubleday, 1997), 37.

27. Campbell-Kelly and Aspray, 181.

28. Koss, 51.

29. Koss, 49.

30. Fitz, 21.

31. Paul Edwards, "The Army and the Microworld: Computers and the Politics of Gender Identity," *Signs* 16, no. 1 (1990): 105, 125; emphasis added.

32. Vannevar Bush describes the action of a stenographer as: "A girl strokes its keys languidly and looks about the room and sometimes at the speaker with a disquieting gaze." Vannevar Bush, "As We May Think," *The Atlantic Monthly,* July 1945, repr. 1994, http://www.ps.uni-sb.de/~duchier/pub/vbush/vbush.txt. Women have been largely displaced from programmers to users, while continuing to dominate the hardware production labor pool even as these jobs have moved across the Pacific.

33. For more on the software crisis and the relationship between it and software engineering, see Cambell-Kelly and William Aspray, 196–203, Ceruzzi, 105, and Frederick P. Brooks's *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition (NY: Addison-Wesley Professional, 1995).

34. Mahoney, 15.

35. Edsger W. Dijkstra, "EWD 1308: What Led to 'Notes on Structure Programming,'" in *Software Pioneers,* ed. Broy and Denert, 342.

36. Edsger W. Dijkstra, "Go to Statement Considered Harmful," in *Software Pioneers,* ed. Broy and Denert, 352, 354.

37. Arguably, go tos are harmful because they are moments in which the computer is directly addressed—conditional branching and loops enable a program to move to another section without such an address. This address poses the question:

exactly who or what is to go to a different section? Intriguingly, go to is a translation of the assembly command "transfer control to"—to give control over the program to the command located in address X. The difference between go to and transfer control to underlies the crisis of agency at programming's core.

38. John V. Guttag, "Abstract Data Types, Then and Now," in *Software Pioneers,* ed. Broy and Denert, 444.

39. Guttag, 444.

40. Guttag, 445.

41. David Eck, *The Most Complex Machine: A Survey of Computers and Computing* (Natick, MA: A. K. Peters, 2000), 329, 238.

42. Koss, "Programming on the Univac 1: A Woman's Account," 49.

43. Paul Edwards, "The Army and the Microworld," 108–9.

44. For more on the pleasure of programming see Linus Torvalds, *Just for Fun: The Story of an Accidental Revolutionary* (New York: HarperBusiness, 2001); and Eben Moglen, "Anarchism Triumphant: Free Software and the Death of Copyright," *First Monday* 4, no. 8 (2 August 1999), http://firstmonday.org/issues/issue4_8/moglen/index.html.

45. See Edwards's discussion of John McCarthy's work in *The Closed World,* 258.

46. John von Neumann, *Papers of John von Neumann on Computing and Computer Theory,* ed. William Aspray and Arthur Burks (Cambridge: MIT Press, 1987), 413.

47. Ben Schneiderman, "Direct Manipulation: A Step Beyond Programming Languages," in *The New Media Reader,* ed. Noah Wardrip-Fruin and Nick Montfort (Cambridge: MIT Press, 2003), 486.

48. Brenda Laurel, *Computers as Theatre* (Reading, MA: Addison-Wesley Publishers, 1991), xviii.

49. Lev Manovich, "Generation Flash," 2002, http://www.manovich.net/DOCS/generation_flash.doc.

50. This notion of empowerment begs the questions of what it means to produce "original code" and how Bauhaus design in the service of information design is not citation. Manovich, "Generation Flash."

51. Manovich, "Generation Flash."

52. Manovich, "Generation Flash."

53. Fredric Jameson, *Postmodernism, or The Cultural Logic of Late Capitalism* (Durham: Duke University Press, 1991), 51.

54. See Wendy Hui Kyong Chun, *Control and Freedom: Power and Paranoia in the Age of Fiber Optic* (Cambridge: MIT Press, 2005).

55. Louis Althusser, "Ideology and Ideological State Apparatuses (Notes Towards an Investigation)," in *Lenin and Philosophy and Other Essays,* trans. Ben Brewster (New York: Monthly Review Press, 2001), 109.

56. See Slavoj Zizek, *The Sublime Object of Ideology* (London: Verso, 1989), 11–53.

57. See Ceruzzi, 208–9; Cambell-Kelly, 207–29.

58. However, these parallels arguably reveal the fact that our understandings of ideology are lacking precisely to the extent that they, like interfaces, rely on a fundamentally theatrical model of behavior.

59. Jacques Lacan, *The Seminar of Jacques Lacan, Book II: The Ego in Freud's Theory and in the Technique of Psychoanalysis* (1954–1955), ed. Jacques-Alain Miller, trans. Sylvana Tomaselli (New York: Norton, 1991), 81.

60. Schwartz, C1.

61. Schwartz, C1.

62. Manovich, *The Language of New Media*, 48.

63. See Media Player's "Licensing Agreement."

64. Microsoft is considering such actions in its Palladium initiative. See Florence Olsen, "Control Issues: Microsoft's plan to improve computer security could set off fight over use of online materials," *Chronicle of Higher Education,* 21 February 2003, http://chronicle.com/free/v49/i24/24a02701.htm.

65. See Kittler, "There Is No Software."

66. See Francois Jacob, *The Logic of Life: A History of Heredity,* trans. Betty E. Spillman (New York: Pantheon Books, 1973), 247–98.

67. Richard Doyle's concept of "rhetorical software," developed in his *On Beyond Living: Rhetorical Transformations of the Life Sciences* (Stanford: Stanford University Press, 1997), epitomizes the use of software as a critical term in non-scientific scholarly discourse and reveals the extent to which software structures our ideas and thus lies as the concept that needs most interrogation.