TRANSPARENT CULTURES: IMAGINED USERS AND THE POLITICS OF SOFTWARE DESIGN (1975-2012)

BY

MICHAEL L. BLACK

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in English
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Robert Markley, Chair
Associate Professor Ted Underwood
Associate Professor Melissa Littlefield
Associate Professor Spencer Schaffner
Associate Professor John T. Newcomb

Abstract

The rapid pace of software's development poses serious challenges for any cultural history of computing. While digital media studies often sidestep historicism, this project asserts that computing's messy, and often hidden, history can be studied using digital tools built to adapt text-mining strategies to the textuality of source code. My project examines the emergence of personal computing, a platform underlying much of digital media studies but that itself has received little attention outside of corporate histories. Using an archive of technical papers, professional journals, popular magazines, and science fiction, I trace the origin of design strategies that led to a largely instrumentalist view of personal computing and elevated "transparent design" to a privileged status. I then apply text-mining tools that I built with this historical context in mind to study source code critically, including those features of applications hidden by transparent design strategies.

This project's first three chapters examine how and why strategies of information hiding shaped consumer software design from the 1980s on. In Chapter 1, I analyze technical literature from the 1970s and 80s to show how cognitive psychologists and computer engineers developed an ideal of transparency that discouraged users from accessing information structures underlying personal computers. Prior to the early 1980s, software designers assumed that users were at least semi-literature in the technical details of computer systems. In response to the expanding popular interest in personal computing, engineers abandoned this assumption and began to design hardware and software for users with little or no understanding of computer systems. Successful software design became defined by the invisibility of computation. Users were made to feel that they were directly manipulating their data. Intimidating command prompts were hidden, and the complex operations performed through them automated behind simpler, visual metaphors like

the desktop. In contrast to its political connotations, “transparency” in the context of design represents a strategy of withholding technical information. As I argue, transparency’s paradoxical promise stems from the instrumentalist belief that problems of access should be addressed through an artificial increase in technological literacy.

Chapter 2 argues that the commercialization of transparent design established a computational horizon. In analyzing popular computing magazines published during the early 1980s, I show how Apple positioned the Macintosh in opposition to the rationalism of IBM, promising that the future of computing would be one in which users would be empowered by machines that “just worked.” Apple established transparency as the teleological purpose of technological development: computational power must remain invisible even as its *representations* become increasingly visual. Yet Steve Jobs was not the only one to imagine such a future. The cyberpunk writings of William Gibson and Bruce Sterling respond to the cultural constraints of transparent design. In cyberpunk’s dystopias, transparency reifies technocratic power, granting only a small number of corporate actors control over computation. While critics have argued that Gibson and Sterling strive to make information systems visible, I argue that their hackers are constrained by transparency, regularly encountering limits to their own computational agency in an ultra-transparent cyberspace. Cyberpunk thus offers a vision of a future where computers “just work,” as Apple promised, but often at cross-purposes with their users.

Even if the veil of transparency were lifted, modern software is constructed according to principles that resist critical readings of source code. Chapter 3 asserts a need for methodologies in software studies that can account for the scale and scope of contemporary software. Whereas early software was written from beginning to end much like a script or an essay describing a

process, more recent languages break software into an assemblage of encapsulated components. As I argue, fears of "spaghetti code"—software with routines so entangled that delineating specific processes is impossible—imagine developers succumbing to the sort of the mania described in Robert Coover's *The Universal Baseball Association*. Object-oriented programming, by contrast, stresses modularity and encourages programmers to write components with singular functions that can be incorporated into software by their peers without needing to be managed directly. In this manner, structured programming's guiding principles produce a hierarchy both within software and through acts of collaborative programming. This "freedom from" overwhelming complexity, however, comes at the cost of producing highly constrained network perspectives that resemble those found in Margaret Atwood's *The Handmaid's Tale*. Accounting for this structure, I conclude, requires something more than the close reading of code's aesthetics advocated by methodologies like Critical Code Studies.

The final two chapters are case studies that build text analysis tools to map software's sprawling assemblages and rapid evolution, respectively. Chapter 4 takes as its object the open source software movement, studying both its various manifestos and the source code for software produced by its members. Examining the movement's foundational texts, I argue that open source principles reflect a strong opposition to transparent design. Its founders assert that writing software is a form of expression and refuse to distinguish between sociocultural and technical concerns. In theory, open source software should reflect these principles; however, by applying topic modeling and social network analysis tools to source code for the Linux operating system, I demonstrate that it exhibits a form of structural obfuscation similar to that in commercial software. This study suggests not only that structured programming relies on the same principles and strategies as proprietary design but also that transparency's obfuscation of computation need

not restrict the user's agency. These restrictions are a deliberate choice by designers rather than an inherent feature of software as a medium. Despite the system's structural obfuscation, Linux's designers account for the political implications of structured programming by allowing users to bypass transparency entirely, leaving core mechanisms open to configuration in ways that Apple and Microsoft forbid.

Chapter 5 takes as its object of study the source code to sixty versions of Mozilla Firefox, tracing the program's ancestry to Netscape's decision to release the source code to its Communicator browser suite in 1998. Including source code in our understanding of software's materiality exposes a complex bibliographic history that quickly becomes too large to study through traditional methods of close reading. In order to address the textual duplication present in a corpus of successive versions of an application, I build topic modeling software tailored to source code's iterative inscription. In expanding software's materiality to include its source code, this case study shows that software's internal structures are subject to constant evolution and that transparent design often leads to large-scale changes in software that go unnoticed by users as long as the interface remains relatively unchanged. While digital tools can help us explore large bodies of cultural data, a critical theory of software construction is essential to help make sense of it.

To my parents, David and Debbie

Acknowledgements

This project would not have been possible without the excellent advice and support I received from my dissertation committee. I'd like to thank Robert Markley, whose guidance and feedback over the past six years has not only made me a better writer but also kept me excited about my scholarship even during its most difficult periods. Thank you to Ted Underwood and Melissa Littlefield for helping me navigate the challenges of interdisciplinary scholarship and for providing me with opportunities to engage with research practices outside of the humanities. Without Spencer Schaffner's encouragement to try building software around my ideas, this project might have looked very different. I'd also like to thank Tim Newcomb, whose questions helped keep me grounded in the humanities. Finally, I'd like to thank Miles Orvell and Susan Wells who both first pushed me to pursue my interest in exploring digital culture in an English department even though it meant having to move across the country to do so.

Table of Contents

Introduction –Thinking Outside the (Black)Box

"None of us science fiction writers foresaw phone freaks. Fortunately, neither did the phone company, which otherwise would have taken over by now."

—Philip K. Dick, from "The Android and the Human" (1972)

"The future is already here—it's just not very evenly distributed."

—William Gibson, from an interview with *Fresh Air* (1999)

In 1968, a blind college student at the University of South Florida named Joe Engressia discovered how to get free long distance telephone calls by whistling at just the right pitch into his phone. As news of his feat spread, so too did the idea that information systems could be empowering rather than simply oppressive. In the years that followed, rogue engineers likes John Draper (a.k.a. "Captain Crunch") covertly circulated blueprints for small audio devices they called "multi-frequency boxes" that allowed enterprising phone hackers to do far more than dodge phone bills. These early hackers boasted in interviews that by holding the box up to the phone and pressing its keys in specific sequences they could listen in on other calls, interrupt someone's service, host intercontinental party calls, or even seize control of whole phone systems by routing calls through a tangle of trunk lines. But as Draper explains, his interest was academic rather than disruptive: "I do it for one reason and one reason only. I'm learning about a system. The phone company is a System. A computer is a System. If I do what I do, it is only to explore a System. Computers. Systems. That's my bag. The phone company is nothing but a computer. . . .Ma Bell is a system I want to explore. It's a beautiful system" (qtd in Rosenbaum

120).[1] While AT&T and its Bell System previously had represented a power so omnipresent that activists warned each other in whispers not to say anything subversive in the same room as a telephone, these "phone freaks" demonstrated that information technology could be appropriated and re-purposed into a creative outlet. But phreak culture did not last long. By 1972, Engressia and Draper had been arrested, and Abbie Hoffman's attempt to draw other phone freaks into the Youth International Party proved disastrous. Nonetheless, a certain fascination with the way the phone freaks were able to reverse engineer a dominant communication technology into something more than an extension of corporate power lingered in popular culture: a belief that if a person is smart enough, he or she could find a way to wrest control of information technology away from corporate technological and economic control and transform it into a creative medium.

The personal computer revolution of the late 1970s, and its rhetoric of individual empowerment via technology, emerges out of phone freak culture both thematically and historically. After seeing one of the earliest video games, Stewart Brand, a champion of the counterculture during the 1960s and 70s, claims in a 1972 article for *Rolling Stone* that even though "a few brilliantly stupid computers" carrying out the orders of a handful of oligarchs with ruthless efficiency "can wreak havoc, a host of modest computers (and some brilliant ones) serving innumerable individual purposes can be healthful, can repair havoc, feed life."[2] Because

[1] For interviews with Engressia, Draper, and other phone freaks, see Maureen Orth, "For Whom Ma Bell Tolls Not." *Los Angeles Times*. Oct 31, 1971 and Ron Rosenbaum, "Secrets of the Little Blue Box." *Esquire* Oct 1971.

[2] Retrieved from http://www.wheels.org/spacewar/stone/rolling_stone.html

of their cost, computers generally were associated only with the large corporations that manufactured them, like IBM and DEC, and the powerful corporate, academic, and government laboratories that used them—often as part of projects funded by military contracts. In popular culture, these machines were extensions of the economic, cultural, and military hegemony that large technocratic bureaucracies pursued. But for Brand, games like *Spacewar!* show that computers themselves are not evil. Video games, like the phone freaks, demonstrate what is possible if computers are able "to come to the people." Throughout the 1970s, groups like the People's Computer Company and the Homebrew Computer Club tried to fulfill the promises that Brand and others described. These hobbyists took advantage of the first commercially available microprocessors, modifying parts that they could order from circuit board manufacturers and cobbling them together in basements, garages, and dorm rooms in pursuit of their own "personal" computers. Among them were Bill Gates, Paul Allen, Steve Jobs, and Steve Wozniack. While the former pair quickly dissociated themselves from the rhetoric of the personal computer revolution by allying themselves with IBM, the latter emerged out of phone freak culture and soon transitioned from covertly selling multi-frequency boxes to friends at Berkeley to marketing the first commercially successful personal computer. Their Apple II system became a symbol of the personal computer revolution. Children in the late 1970s and 80s were introduced to it at school as a source of fun and intellectual exploration, whether drawing with a LOGO interpreter or playing educational games like Odell Lake and The Oregon Trail. Apple has since retained, at least nominally, its countercultural agenda and its rhetoric of self-empowerment through technology by positioning its Macintosh line of machines, smart phones, tablets, and portable music players as "computers for the rest of us" and by encouraging its customers to "think different."

What distinguishes the countercultural ethos of Apple and other personal computer manufacturers from that of the phone freaks are the companies' efforts to monopolize technical knowledge in order to protect proprietary software systems. While the phone freaks, and hacker culture in general, revel in the free-form probing of complex software and hardware systems, personal computers today are designed as appliances: blackboxes that perform specific creative, productive, and leisure activities but reveal little about their inner workings. Companies like Apple and Microsoft purposefully design their personal computer hardware and software to resist the playful tampering celebrated by the phone freaks and early computer enthusiasts. Commercial developers assume that the average user is unwilling or incapable of learning about computer systems, victim to a learned helplessness. Instead of trying to education consumers, however, commercial models opt to provide their users with a heavily sanitized and simplified vision of computing, one which leaves corporations like Apple, Microsoft, and Google as the sole arbiters of the roles that computational systems should play into modern society.

This dissertation examines the history and implications of a theory of usability that human-computer interaction theorists and commercial software designers in the 1980s dubbed "transparent design." This conceptualization of transparency is paradoxical: it promises access to computing power by denying direct access to the computational processes that structure that power. Transparent design is a highly mediated practice, blending J. David Bolter and Richard Grusin's new media dialectic of hypermediation and immediacy into a user-friendly interface.[3] While transparent design is often presented as a way to "simplifying" computing, programmers must employ increasingly sophisticated techniques of hypermediation in order to fulfil

[3] See Bolter and Grusin's *Remediation: Understanding New Media* (2002).

transparency's promise of direct manipulation: the experience by users of an immediate relationship between themselves and the realties represented in their data. The initial love affair between Silicon Valley evangelists and virtual reality technology during the mid-1980s and early 1990s unsurprisingly coincides with the first commercially available transparent personal computer systems. Like virtual reality systems, a transparent operating system such as Apple's MacOS or Microsoft's Windows "erases itself, so that the user is no longer aware of confronting a medium, but instead stands in as an immediate relationship to the contents of that medium" (Bolter and Grusin 24). Software that emphasizes an experience of direct manipulation also discourages any acknowledgement of the mediating processes that produce that experience. Transparency's influence over computational culture has only increased since the release of the Macintosh in 1984, effectively becoming the singular guiding principle of software and hardware design. We can see transparency's influence in every cultural sphere that computers touch: in representations of computers in the work place, in the home, and in the fantastic futures of science fiction literature and film, as well as in the new models of instant, on-demand media access and the semi-intelligent database agents that anticipate our search needs. Transparent computers can fulfill all our needs, their designers promise. Consumers no longer need to consider technical specifications or concern themselves with compatibility, they just need to hear Jeff Goldblum explain that an Apple computer will get you into the party that everyone else is having on the Internet within 10 minutes or see that Microsoft's Surface tablets make work seem so much fun that users break out into a dance in the conference room. But the fine degree of proprietary control that transparent design allows commercial developers to exert over our modes of computer use also forbids the free-form exploration of systems that characterized the early periods of personal computing. Even though the image of computing has softened from the

digital dystopia of highly centralized, mainframe supercomputers, transparent design ensures that computer power remains carefully controlled, directly accessible only the corporations, technical labs, and government institutions that design and implement software systems.

Transparent design initially emerged as the solution to a problem of technological literacy. By the late 1970s, the promises of personal computers were well known, but the act of using a computer was still frustrating. Robert Cowen's sobering account of the realities of the personal computer revolution in a 1981 issue of *The Technology Review* complains that "merely to use computers to do things we already do with less expense and effort would be foolish. Yet this is about all that has been offered so far. . . .Friends sometimes ask if I had put the family budget on the computer, but I haven't because I can easily keep track of it with a pocket calculator. Why glorify the trivial?" (6). Because very few of a computer's processes were automated, users had to be directly responsible for managing their data-encoding processes, a requirement that made something as simple as inscribing and retrieving recipes from a disk tedious compared to pulling out a box of cards. Using a computer prior to the rise of transparent design required an advanced degree of technological literacy. Users had to memorize commands, know which directories to stay out of lest they risk damaging their operating system, and even know how to do a bit of programming themselves if the software for the task they wanted to undertake was not readily available. In response to these problems, cognitive psychologists and computer engineers like Donald Norman, Ben Shneiderman, Terry Winograd, and Fernando Flores proposed that interface design strategies should appeal less to higher level cognitive functions like linguistic memory and instead represent systems in simpler, subtler terms using

shapes, colors, and symbolic patterns that suggested a non-arbitrary relationship to function.[4] Yet doing so required automating those parts of computer systems that could not be easily represented with visual metaphors. Transparent design changed how we imagine the act of computing: technological literacy today is understood not as the ability to decipher the intricacies of software architecture or trace patterns in networked systems but is instead usually framed in terms that emphasize the *use* of technology: "an understanding of technology at a level that enables effective functioning in a modern technological society" (Gamire and Pearson 2). The contemporary vision of a modern technological society is not one comprised solely of software and electrical engineers. Rather, today we live out a *Star Trek*-like fantasy with lavish touch screens, wearable devices, and hand-held libraries; all users need to know to participate in this fantasy is how to read highly stylized interactive displays.

From the perspective of human-computer interaction studies, a technological object is transparent when its users no longer need to acknowledge directly its presence when they undertake a task. Transparent technology should be understood by its users "intuitively" through a seamless blend of visual, audio, and/or haptic cues that serve as metaphors for familiar concepts or that users can internalize readily. Several examples of easy to use, transparent technologies and confusing usability failures can be found in Norman's *The Design of Everyday*

[4] Norman, Shneiderman, Winograd, and Flores are all authors of influential texts in the field of human-computer interaction studies. See Norman's *The Design of Everyday Things* (1988), Shneiderman's "Direct Manipulation: A Step Beyond Programming Languages" (1983), and Winograd and Flores' *Understanding Computers and Cognition: A New Foundation for Design* (1987).

Things (1988), a foundational handbook featured in most human-computer interaction curricula. Phones, for instance, can be both easy and difficult to use. The shape of the phone suggests placement alongside the ear and mouth, with audio coming from handset speaker signaling to users if they need to correct its orientation. Buttons are clearly labeled with numbers, and pressing them produces a tone that, while primarily aimed at the phone system's routers, serves the important secondary purpose of indicating to users that their input was received. More complex phones can be tricky, however, especially if the features like hold/resume, transfer, switch lines, speaker phone, and mute are not clearly labeled or require more than a few key presses. A complex array of buttons with small labels or a limited set of function keys that must be pressed in sequence to carry out call operations can pose problems unless a user has spent significant time learning how to operate them. Today's smartphones are far more complicated than the phones Norman examines in the 1980s, and yet their touchscreen and LED software interfaces allow for these operations, as well as new ones not possible on older phones, to be carried out swiftly by just about anyone. Users do not have to pause and review a chart, read a manual, or pour over a dozen or more small buttons. Their software interfaces have buttons on screen that change functions while at the same time clearly identifying what operations they will perform when pressed. Transparent design is, in this sense, a strategy of making tasks increasing visualized while at the same time suppressing technical information. Users don't need to know much about the systems they use because software now automatically manages itself, selecting and presenting users with sets of options based on its observations of their behavior.

Narratives of personal empowerment through the use of personal computers and related devices are thus predicated upon a surface reading of technology. Today, anyone can purchase a laptop computer, remove it from the box, plug it in, and immediately begin using it after entering

in a bit of personal information to register his or her identity on the machine. This hypothetical laptop owner doesn't need to know much at all about the computer's operating system software, except perhaps where to locate other programs he or she may install—or, more accurately, that install themselves at the user's request—and where to find a list of wireless hotspots. Video game consoles represent this fantasy of readily available technological power even more dramatically because users need only insert a disc or cartridge and press a button to be transported almost instantly to highly stylized virtual worlds. Pressing a button or nudging a thumbstick leads to instant audio and visual feedback, increasingly accompanied by a rich musical score and controller vibrations to add a sense of dramatic impact to the player's decisions.

Considered as part of a narrative of innovation and technological progress, the degree of transparency in personal computers and gaming systems is almost sublime. Machines more powerful than a mainframe supercomputer from the 1950s or 60s became available for purchase in stores in the 1980s; and, more dramatically, the phones we carry in our pockets rival the power of early big iron mainframes and are able to download in seconds small "apps" that would have required thousands of dollars of disks just to store prior to the mid-1990s. Now, our personal computing devices turn on in less than a minute, operate silently, and provide immediate access to algorithmic operations so complex that they hardly could have been imagined several decades ago. Yet at the same time that computers have become so highly visualized, self-representational, and easy to access, their inner mechanisms have become increasingly invisible. Computers are no longer intimidating to users because the vastness of the resources they contain are hidden behind interfaces that present users with intuitive metaphors rather than detailed system information. We now live in a culture where access to technological

power appears readily available, but we cannot see how the software that structures that power channels our use of it towards and away from certain ends. Because we're trained by technological interfaces to think about the act of computing in non-computational terms, users may not be able even to imagine those possibilities that transparent design encourages us to ignore.

Because transparency constrains our ability to perceive and articulate algorithmic processes, its selective obfuscation has important implications for digital media studies, the digital humanities, and other fields taking a critical approach to the history of information technology. Digital media critics and historians, in short, are users too and thus subject to the same discursive constraints resulting from transparent design. Although transparent design is celebrated by Silicon Valley historians and fawned over by hordes of fans hungry for the next even easier to use version of their favorite game or device, there is an unacknowledged political dimension cementing its commercial and ideological ubiquity: transparent design essentially splits technology around two distinct models of engagement. There are those who have access to technical information about how computers work and those who have access only to instructions on how to use them. Unable to develop an advanced degree of technological literacy without specialized training at universities or within exclusive professional circles, users become more and more reliant on transparency's strategy of simplification through obfuscation. William Gibson's pronouncement that "the future is already here—it's just not very evenly distributed" reflects this digital divide, but in ways that cannot be explained fully in terms of racial, economic, or gender stratifications. These issues are still implicated in software design, and Anne Balsamo, Edmond Chang, Alexander Galloway, Lisa Nakamura, Cynthia Selfe, and Judy Wajcman, among others, have shown that decisions about interface design are wrapped up in

often unacknowledged assumptions about race, class, and gender, and sexuality. Yet it is very difficult to see the full scope of those assumptions. We can try to interpret the metaphors that designers employ to represent computational operations to us, but we can only see the vehicle and not the tenor. Transparency conceals from us the layered architecture of software, of the complex information networks of components that support and render user-friendly interfaces. It is difficult to understand how developers' assumptions about the identity of users and their information habits play out beneath the interface. Transparent design, despite the rhetoric of direct manipulation and immediate experience data, prevents us from directly accessing computational knowledge. Hacker narratives like *The Matrix* or *Tron* would have us believe that anyone with enough skill can look at software and see into its code, recognizing the presence of other people, objects, and processes just as readily as they would in other contexts, but transparency makes that future impossible. Under the regime of transparent design, the world of code is shut off from us, enfolded by the transformational processes that turn a programmer's source code into machine readable binary sequences and further protected by copyright and patent laws. As critics, historians, and users of software, we not only need to understand how our methodological approaches are shaped by transparent design but also how to develop both techniques that explore the processes it hides and critical languages that can articulate the sociocultural implications of technical specifications.

Science fiction literature, perhaps, in part, because it cannot match film's visual celebration of technology, is adept at considering the implications of a suppressed world of textual codes. In a 1988 interview with *Locus* magazine, the influential cyberpunk writer Bruce Sterling argued that the goal of science fiction following the personal computer revolution should be to address "the threatening technological innovations" that surround us and act on us

invisibly and continually, "mak[ing] us feel like straws in the wind" (73). If the impacts of Cold War era technology are readily imaginable, thanks to a deluge of visions of nuclear holocaust in television and film, the effects of information technology are more felt than seen. Science fiction, in other words, serves as a platform for exposing those effects by imagining and reflecting on our experience of them. In 1988, Gibson commented that even though he had little knowledge of how computers work, he nonetheless writes fiction with a goal of making their unseen operations more visible to his readers.[5] Even if Gibson, like us, has a perspective bound by the technology of the late 1980s and sometimes misconstrues their hidden operations for narrative effect, his novels and those by other authors influenced by his work help provide readers with a vocabulary to articulate their experience of the politics of transparency in ways that film can't or won't. Novelists like Margaret Atwood, Greg Bear, Octavia Butler, Pat Cadigan, Cory Doctorow, Rudy Rucker, Neal Stephenson, and Charlie Stross, regardless of their relationship to the cyberpunk movement, have all since imagined both dystopian and utopian future societies where life is entangled within complex networks of information and every action is observed, discretized, categorized, constructed, or responded to according to sets of algorithmic processes, both digital and biological.

By positioning science fiction as a critical tool for the cultural study of science and technology, I assume that fiction is already in direct conversation with technical literature. This approach is distinct from other critical works that view fiction as distinct from and/or in opposition to technoscience. Joseph Tabbi, for example, argues that literature is a valuable critical tool because literary writers possess an "outsider perspective" that allows them to

[5] See his interview with Larry McCafferty in *The Mississippi Review* 16.2/3 (1988).

represent “not the technology itself but the tumultuous and incongruous nature of postmodern experience” (25). For Tabbi, science fiction is valuable in the way that it distances us from the technical specifics emphasized exclusively in professionalized, technoscientific genres. Yet there is a risk in maintaining the separation between the technological and the literary that Tabbi defends. Not only does he imply that there is a clean separation between science fiction and technoscientific practice, but he also assumes that the technical specifics that the literary eschews are of little value to cultural studies of science and technology. Like science fiction, technical literature actively imagines future contexts of computational use and implementation. The primary difference between these genres is thus one of temporal scale, with technical literature focused more on the immediate effects and fiction on the longer term, more abstract consequences of design decisions. In this respect, this project follows models of scholarship put forth by Balsamo, N. Katherine Hayles, and Robert Markley, to name a few.

Specifically, studying the discourse taking place in white papers, technical journals, computer science textbooks, programming manuals, and science fiction that surrounds personal computing can help us to understand not only how and why transparency came to dominate contemporary software design but also which possible futures were cast aside in favor of it. Because commercial software developers must attract and retain new consumers for their products, they often present design choices to us as natural or intuitive. A new version is more “efficient,” “elegant,” “powerful,” or “user-friendly” than the last. These terms all imply a natural progression and essential principles of human-machine relationships that strategies of transparent design push us further towards. The commercial success of transparent design has, in other words, changed transparency from simply a means of making computers accessible to one of the primary goals of software design. But as Lawrence Lessig reminds us in *The Future of*

Ideas, "Cyberspace has no nature. How it is—what barriers there are—is a function of its design, its code. . .it makes no sense to speak about the nature of this system that is wholly designed by man. Its nature is as man designs it" (121). Throughout his body of work, Lessig repeatedly argues that computer systems enact the will of their designers, producing algorithms that govern the behavior of users in ways that mirror, and often supersede, the coercive power of law. If personal computers are really to become more than just appliances that encourage media consumption, we need to recognize that because our relationship to them is not natural. We do not need to view transparency as the "ideal" or even "best we can do" in terms of designing software systems that are accessible and empowering. In her "Cyborg Manifesto," Donna Haraway argues that the "machine is not an *it* to be animated, worshipped, and dominated. The machine is us, our processes, an aspect of our embodiment. We can be responsible for machines; they do not dominate or threaten us" (180). Although Haraway is concerned primarily with resisting the way technoscience polices women's bodies by attempting to naturalize its relationship to them, her point that "the illegitimate offspring of militarism and patriarchal capitalism. . .are often exceedingly unfaithful to their origins" rings true for personal computing (151). It is important to recognize that principles considered inherent or essential to software are neither because software and our relationship to it is wholly socially constructed. What we should fear is the essentialist rhetoric used to keep technocratic power in place, not software itself. That transparency remains the dominant mode of design and establishes a singular model of "user-friendliness" speaks more to the entrenched power of developers like Apple and Microsoft than to any essential truth it represents about human-machine relationships. Computers can be anything we want them to be, and if we are able to study, criticize, and reshape their algorithms to our own ends, we can redefine what it means to be user-friendly in ways that

appeal to imagined futures other than those proposed by corporate developers.

While studying software's paratexts is useful primarily for interrogating technological reasoning, it still leaves us a step removed from algorithmic processes. The idea that software is not fully or even accurately represented on screen is not new to digital media studies. Lev Manovich, for example, proposes that new media objects are multi-layered, split between a "cultural" and "computational layer." The cultural layer represents those elements users see on screen, the product of transcoding algorithms that operate at the computational layer. It is tempting, as Alexander Galloway does, to explore this multilayered structural operation as an analogue to Althusserian ideological state apparatus or Marxist false consciousness. Transparent software does indeed require many rituals from its users, hailing them with an array of defaults that most users casually accept without a second—or even a first—thought, and its collection of signifiers reflect a model of individuality that obfuscates its corporate origins and one-size-fits all model of usability. Similarly, transparent design bears a striking similarity to the hegemonizing effect of mass culture that Max Horkheimer and Theodore Adorno describe, yet it is not concerned with imagining how computers work: transparent design discourages users from imagining algorithmic processes at all. While the following chapters do draw on and extend these analytics, I believe we must resist the temptation to map familiar models too readily onto the deeper structures of software hidden by transparent design. Mapping these complex heuristics onto only those signifiers visible on the surface of a transparent interface can help us to describe the experience of alienation from those elements that transparent design suppress. But they do not solve the problem of access, nor is there any guarantee that they will map readily on deep structures of software described in source code.

This project is therefore not only a cultural history of transparent design but also a call

for a critical language that acknowledges the distinct textual qualities of source code. Because the interface obfuscates the internal structure of software during runtime, reading source code is the best way to analyze those algorithms that transparency hides. The impulse to develop such a language has long been a part of digital media studies; however, critics have grown so accustomed to transparency as a discursive filter that they rarely acknowledge the complex and hidden bibliographical practices that produce software via source code. Specifically, we need to address the ways in which object-oriented programming, by far the dominant style of source code organization, resists close readings of code and supports coordinated inscription on a scale that rivals and often surpasses the size of textual archives studied in the humanities. While digital media studies first wrestled with a language of procedurality, later works by Ian Bogost, Matthew Fuller, Katherine Hayles, Matthew Kirschenbaum, Mark C. Marino, Lev Manovich, Nick Montfort, and Noah Wardrip-Fruin push us to examine the structure of source code itself. This approach is distinct from scholarship influenced by German media theorist Friedrich Kittler who argues that "there is no software" in the sense that all the stuff we call software is always already structured by, and thus inseparable from, the hardware platforms on which it exists. Wendy Chun, for example, develops Kittler's point by arguing that we must "engag[e] its odd materializations and visualizations closely and refus[e] to reduce software to codes and algorithms—readily readable objects—by grappling with its simultaneous ambiguity and specificity" (11). Although Chun is persuasive in her argument that source code may not reflect the experience of using software or be too abstract in its description of use-case scenarios to reflect much about its contexts of implementation, source code is nonetheless an imagined description of software function, of potential human-machine relationships, and assumptions about the information cultures in which software participates. Studying source code is thus a

necessary complement to, but not a replacement for, the rich body of scholarship in digital media studies written from a user's perspective. Reading source code critically requires an analytic methodology informed by the concerns of critical and textual theory as well the technical language of computer science: a method that can articulate the way software's inscription is influenced both by technical standards and the larger cultural dynamics in which both its developers and users participate.

Finally, I must acknowledge that not everyone in digital media studies and the digital humanities wants to develop the advanced technical literary required to read and write source code. Nor should they have to in order to make strong contributions to the critical study of digital culture or the application of text analytics to the study of literary aesthetics. At this project's core is an argument that not just humanists, but a general public of users, must be able to engage with technology in ways beyond the superficial model that commercial applications of transparent design encourage. The Free and Open Source Software communities demonstrate that not every person who uses software needs to read its source code to promote an open discussion of its sociocultural implications, just as not everyone needs to follow the minute details of court proceedings to discuss the potential consequences of judges' decisions. So long as the information is publically available, accessible not just by a small group of technocrats, concerned citizens can read and share problematic details with others. At its core, this project argues that the opportunity for such a discussion must be available and can only be made available if source code is accessible. Although software is treated as intellectual property by law, many types of software like operating systems, network architecture, web browsers, and cloud data management services function as social infrastructure. Transparent design is not problematic because it makes software easier to use but because it does so in a way that exploits the

illegibility of complex algorithmic processes in order to produce a firm divide between those who are able to see and try to understand those processes and those who can only be subject to them. If we can learn to think outside the discursive constraints of transparent design, we can propose new models of usability that blend ease of use with the freedom to study, to critique, and to modify software. So long as we continue under an ideology of transparent design, we are not only subject to the assumptions about digital culture that commercial software developers make for us—that everyone who wants to listen to music or watch movies over the Internet is a potential media pirate or that allowing the NSA to monitor our computational activities via backdoors is good for democracy—but we may well have to face the problem of large portions of digital history remaining forever inaccessible. If the structures that transparency hides are never analyzed, source code will remain forever invisible, even after the software built from it becomes obsolete, because its developers decided that its value is primarily industrial and commercial rather than cultural.

With these ideas in mind, the first three chapters of this project examine how and why transparent design's strategies of information hiding reshaped commercial software from the 1980s on. In Chapter One, I analyze technical literature from the 1970s and 80s to show how cognitive psychologists and computer engineers developed an ideal of transparency that discouraged users from accessing information structures underlying personal computer operating systems. Prior to the early 1980s, software designers assumed that users were at least semi-literate in the technical details of computer systems. Before personal computers "came to the people," they were developed largely for specialist users for specific tasks, and developers thus assumed a degree of cultural and cognitive homogeneity among them. In response to the expanding popular interest in personal computing, engineers abandoned this assumption and

began to design hardware and software for users with little or no understanding of computer systems. Researchers like Donald Norman, Ben Shneiderman, Lucy Suchman, Terry Winograd, and Fernando Flores argued that a perfectly uniform understanding of software across a body of users was impossible.[6] Whereas theories of user interface design had previously defined usability as a problem of human-machine information processing efficiency, transparent design assumes an imperfect user who had to rely on non-technical models to interpret systems. Successful software design thus became defined by the invisibility of computation and the representation of computer systems via metaphors and visual icons. Intimidating command prompts were hidden, and the complex operations performed condensed into icons peppered over metaphoric representations of virtual environments like the desktop. Users were thus made to feel that they were directly manipulating their data through a combination of automation and selective representation. In contrast to its political connotations, "transparency" in the context of software design represents a strategy of withholding technical information. As I argue in this chapter, transparency's paradoxical promise stems from the instrumentalist belief that problems of access should be addressed through a prosthetic substitution for technological literacy.

Chapter Two argues that the commercialization of transparent design establishes a computational horizon. Although transparency was theorized by academic researchers, and prototyped by development think-tanks like Xerox's PARC laboratory, it was codified by commercial software developers. In analyzing popular computing magazines published during

[6] In addition to the work from Norman, Shneiderman, Winograd, and Flores noted above, see also Lucy Suchman's *Plans and Situated Actions: The Problem of Human-Machine Communication* (1987).

the early 1980s, I show how Apple positioned the Macintosh in opposition to the rationalism of IBM, promising that the future of computing would be one in which users would be empowered by machines that "just worked." In doing so, Apple established transparency as the teleological purpose of technological development: computational power must remain invisible even as its *representations* become increasingly visual. Apple's unified model of computer use exploits transparent design to turn personal computers into appliances. When read against IBM's unexpected decision to publish its system documentation and share the technical specifications of its hardware, Apple's description of the Macintosh in interviews and advertisements comes to resemble the Orwellian fantasy it used to attack its competitors. Yet Steve Jobs was not the only one to imagine a future of blackboxed computing. The cyberpunk writings of William Gibson and Bruce Sterling respond to the cultural constraints of transparent design by offering a highly stylized yet technologically illegible future. In cyberpunk's dystopias, transparency reifies technocratic power, granting only a small number of corporate actors control over computation. While critics have argued that Gibson and Sterling strive to make information systems visible, I argue that their hackers are constrained by transparency, regularly encountering limits to their own computational agency in an ultra-transparent cyberspace. Many of the tensest moments in Gibson's Sprawl Trilogy unfold amidst conflicts between humans and highly automated, intuitively usable, yet difficult to trust software agents. Cyberpunk thus offers a vision of a future where computers "just work," as Apple promised, but often at cross-purposes with their users' intentions and desires.

Even if the veil of transparency were lifted, modern software is constructed according to principles that resist critical readings of source code. Chapter Three asserts a need for methodologies in software studies that can account for the scale and scope of contemporary

software. Whereas early software was written from beginning to end much like a script or an essay describing a process, more recent languages break software into an assemblage of encapsulated components. The object-oriented paradigm that dominates modern software application development emerged in the 1980s out a series of responses to the software crisis of the 1960s. When companies like IBM began to switch from a made-to-order model to mass production of computer systems, their programmers struggled to keep up with the increased scale required of flexible, general purpose computer software. As I argue, early styles of programming privilege a cybernetic human-machine analogy, understanding that "good" software development was the product of thinking like a Turing machine. Using Brian Rotman's tripartite model of mathematical signification, I show how the software crisis exposes a dis-analogy between human and machine cognition. Because humans and machines interpret source code differently, "spaghetti code"—software with routines so entangled that delineating specific processes is impossible—poses no problems of legibility for machines but leaves human programmers unable to trace the effects of their work. Robert Coover's *The Universal Baseball Association* participates in the fear that limitations of human cognition will pose serious challenges for programming, highlighting the cybernetic dimension of the software crisis even as engineering writing at that time tried to sublimate it. Coover's novel also helps us to see how the solutions to the software ware are cybernetic even though it largely precedes the rise of object-oriented programming. Margaret Atwood's *The Handmaid's Tale*, written during the rise of object-oriented programming, presents a dystopian vision of a "freedom from" overwhelming complexity; however, it is one that comes at the cost of producing highly constrained network perspectives. Atwood's novel resonates with the way the cybernetic theories of Herbert Simon, particularly his idea of "bounded rationality," came to inform industrial design, and allows us to

see how they eventually influenced the modular encapsulation of object-oriented programming. By locating system intelligence in the rules that govern data transaction between software components, object-oriented programming creates a recursive network of microcosmic computer objects that reclaims the idealized human-machine analogy that the software crisis forced many to abandon. In this sense, object-oriented programming solves the problem of source code illegibility, by offering programmers a trade-off. By compartmentalizing the system so that it is structured as a network of encapsulated modules, programmers can choose to master their understanding of individual components or accept that they will never be able to understand all of them and instead adopt an abstract perspective of component relationships. Accounting for the way this structure hides knowledge within networks of source code documents, I conclude, requires something more than the close reading of code's aesthetics advocated by methodologies like Critical Code Studies.

The final two chapters of the dissertation are case studies that build text analysis tools to map software's sprawling assemblages and rapid evolution, respectively. Chapter 4 takes as its object the open source software movement, studying both its various manifestos and the source code for software produced by its members. Examining the movement's foundational texts, I argue that open source principles reflect a strong opposition to transparent design. Its founders assert that writing software is a form of expression and refuse to distinguish between sociocultural and technical concerns. In theory, open source software should reflect these principles; however, by applying topic modeling and social network analysis tools to source code for the GNU/Linux operating system, I demonstrate that GNU/Linux exhibits a form of structural obfuscation similar to that expected of commercial software. This study suggests not only that programming in free software community relies on the same principles and strategies as

proprietary design but also that transparency's obfuscation of computation need not restrict the user's agency. These restrictions are a deliberate choice by designers rather than an inherent feature of software as a medium. Despite the system's structural obfuscation, GNU/Linux's designers account for the political implications of structured programming by allowing users to bypass transparency entirely, leaving core mechanisms open to configuration in ways that Apple and Microsoft forbid.

Chapter Five takes as its object of study the source code to sixty versions of Mozilla Firefox, tracing the program's ancestry to Netscape's decision to release the source code to its Communicator browser suite in 1998. Including source code in our understanding of software's materiality exposes a complex bibliographic history that quickly becomes too large to study through traditional methods of close reading. Even a single version of a complex software application like a web browser is comprised of hundreds of thousands of lines of code. The task of critically editing two versions to understand the scope of the changes between them would already prove too much for a human reader. Multiple versions of such applications therefore defy conventional analysis. Yet many text-mining methods in use today by digital humanists assume a high degree of difference between texts. In order to address the textual duplication present in a corpus of successive versions of an application, I built topic-modeling software tailored to source code's iterative inscription. In expanding software's materiality to include its source code, this case study shows that software's internal structures are subject to constant evolution and that transparent design often leads to large-scale changes in software that go unnoticed by users as long as the interface remains relatively unchanged. Together, these case studies show that the problems of scale that object-oriented programming hides can be overcome by critics and historians of digital media using topic mining and similar analytics. Yet the results still require

interpretation in the cultural vein that I offer in the first three chapters as well as close attention to software's historical contexts of production. My project ultimately demonstrates that while digital tools can help us explore large bodies of cultural data, a critical theory of software construction is essential to help make sense of this data.

Chapter 1 – Making Computers Disappear: On the Origins of Transparency

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. . . . We are therefore trying to conceive a new way of thinking about computers, one that takes into account the human world and allows the computers themselves to vanish into the background."

—Mark Weiser, former head of the Computer Science Laboratory and Chief Scientist at Xerox Palo Alto Research Center

Software's inherent invisibly is so readily accepted in the cultural study of digital media that to point it out once more seems almost a cliché. And yet most of the projects in the digital humanities that aim to examine and theorize about the "stuff" of software, at least in some part, seek to uncover its cultural forms. Developing an algorithm often requires programmers to devote significant time to making its operations visible to ensure that the instructions described in source code are performed as expected. Although many integrated-development environments now offer tools to identify and track elements of software during runtime, the persistence of command line debugging flags, detailed textual log files, and automatically generated error reports in an era of highly visualized interfaces are reminders of just how much time and effort are still devoted to making software's operations visible. That the archetypal "Hello World!" algorithm is the first software that most novice programmers are asked to write is in this regard no surprise. Variables can be declared and initialized, object relationships carefully structured, and loops run infinitely all without any noticeable trace. Until programmers learn how to print a simple string to a command line interface (CLI), there is no way for them to trace the structure of

the software they write. Even when one is writing software with a graphical user interface (GUI), the challenge remains largely the same—the only difference is the wealth of new tools available to make algorithmic processes visible. Users, too, benefit from software that is designed to be as visible as possible. Today, commercial software designers use a variety of skeuomorphic representations of desktops, folders, documents, paintbrushes, calendar pages, filmstrips, and equalizer sliders to represent software's processes in terms of older, embodied technologies. Complex computational processes are so commonly remediated using the signifiers of older media that we rarely stop to consider what lies beneath those familiar surface representations.

Early computer interface design in government, corporate, and academic laboratories produced prototype GUIs that initially helped users manage defense systems and would later play into utopian narratives of technological empowerment that drove the personal computer revolution.[1] Beginning in 1984, GUIs became part of the corporate strategies of hardware and software firms like Apple and Microsoft to seize control of a chaotic marketplace. Apple's Macintosh, in particular, represents a synthesis of early proposals in usability theory from influential researchers in computer science such as Alan Kay, Donald A. Norman, Ben Shneiderman, and Terry Winograd. According to their theories, making computer software engaging and giving users a feeling of direct manipulation paradoxically requires an intermediary layer that translates the underlying system into an intuitively understood visual

[1] See for example the SAGE Radar System, which first prototyped a GUI interface in 1955, or Douglas Engelbart's "Mother of All Demos", which introduced the desktop metaphor in 1968 along with the promise that personal computers would "augment human intellect" of everyday individuals.

iconography. These design practices, often referred to as "transparency," today comprise the dominant strategy of interface design in the personal computer and consumer electronics industries. Although it's difficult to discern precisely where the term was first used, "transparency" is found as far back as 1965 in engineering journals and IBM reference manuals. Originally used to refer to a practice of data formatting, "data transparency" is achieved when data is stored in such a way that its structure can be assumed, allowing multiple components of a program to pass data quickly among themselves without having to dedicate algorithms to discerning its format.[2] If data is transparent, then individual components can read it without needing to know from where it originated from nor what other components interact with it. In this sense, data transparency simplifies information management in complex systems by privileging immediate contexts. Components only process the data as received, with little regard for others. Even though this use of the term originates in a purely machinic context, the term

[2] Some examples of the term from the Association for Computing Machinery's publication archives include: S. Gorn's "Transparent-Mode Control Procedures for Data Communication, Using the American Code Standard for Information Exchange—A Tutorial," *Communications of the ACM* 8.4 (1965); Carl E. Krebs, C. Bumgardner, and T. Northwood's "Terminal Transparent Display Language," *AFIPS'76 Proceedings*; Keith A. Lantz and Richard F. Rashid's "Virtual Terminal Management in a Multiple Process Environment," *SOSP '79 Proceedings*; and Andrew S. Tanenbaum and Robbert Van Renesse's "Distributed Operating Systems," *Computing Surveys* 17.4 (1985). These examples were chosen primarily to provide a sense of the term's lifespan rather than to suggest some sort of canon in computer science or human-computer interaction studies.

slipped easily into the context of human-machine interaction by the mid-1980s. Instead of stating that "the data is transparent," engineers began to use phrases like "the system is transparent to the user," signifying that users could transfer prior knowledge and experience into that of the software they were using. At the same time, this transference also carries with it the idea of a shallow context, making the complex algorithmic structures that produce software irrelevant for users. Only the representations immediately visible on screen are important.

Transparency in user-interface design is thus similarly a response to the difficulty users faced in managing increasingly complex computer systems. J. David Bolter and Richard Grusin's foundational study, *Remediation: Understanding New Media* (1999), suggests that the effect of transparency is produced via the interplay of immediacy and hypermediacy, the dialectical logics underlying digital media. Bolter and Grusin note that the ideal of a transparent interface is "one that erases itself, so that the user is no longer aware of confronting a medium, but instead stands in as an immediate relationship to the contents of that medium" (23-24). Hypermediacy, on the other hand, is the proliferation of representations, "overlapping or nested windows" that "create a heterogeneous space" within which ideas "compete for the [users'] attention" and which remind users that they are operating in a computation space (32). If transparency encourages users to ignore computation, hypermediacy leads users to celebrate it. Bolter and Grusin observe quite correctly that a sense of transparency in modern GUIs is achieved through the complex algorithmic processes underlying hypermediacy. The desktop environments of Apple's OSX and Microsoft's Windows, for example, weave together a variety of visual metaphors, textual prompts, and audio cues. From this perspective, transparent design strategies are "in fact creating a more complex system in which iconic and arbitrary forms of representation interact" (32). Transparency is therefore achieved more through the quality of the

interactivity that hypermediacy affords than through the accuracy of a GUI's representation of systemic processes. Yet computers have always been interactive. Transparency, as a design strategy, aims to increase the users' subjective sense of software's interactivity by automating computation so that users can concern themselves primarily with tasks which can be understood independently of computation. Computers become "transparent" when their operations are no longer part of users' cognitive engagement with computing. Ideally, users will not stop to think about how to do something; they'll just point and click.

Contrary to the political connotations of the term, transparency in design is achieved by making visible certain computational operations while obfuscating others in the interest of cognitive simplicity. As described by Donald Norman in *The Design of Everyday Things* (1988), the goal of a user-centered design should be to lower the amount of information users must passively learn and actively recall during use: to shift the burden from knowledge-in-the-head to knowledge-in-the-world. In other words, computer interfaces should "take on the appearance of the task" at hand; the system itself can then "disappear behind a façade" that lets users operate "without fear of failure or [of] damag[ing]" the system (183). Representing computation through a readily recognizable iconography does not actually decrease a system's complexity. The processes users initiate and interact with remain dependent upon unseen operations that are managed by operating system (OS) software and are conceptually compressed into visual metaphors. Transparent interfaces are in this sense necessarily incomplete representations: confusing complexities are repressed so that users can operate their computers safely, efficiently, and without concern for the system itself. Recent studies in the digital humanities have already begun to highlight the ideological and governmental structure of software. Alexander Galloway's *Protocol: How Control Exists After Decentralization* (2006) and Wendy Chun's *Programmed*

Visions: Software and Memory (2011), emphasize the underlying automation required to sustain the seamlessness of transparent interfaces. Chun, in particular, notes that transparent interfaces reify a "certain logic of governmentality. . . .offer[ing] us a form of mapping, of storing files central to our seemingly sovereign—empowered—subjectivity" (9). Throughout prolonged interaction, she argues, computers inculcate in users a neoliberal subjectivity that encourages them to desire mastery over the informational environment while remaining ignorant of the algorithms acting on their behalf. While I do not want to imply that OS software and other system management tools are necessarily oppressive, it is extremely important that we understand the nature of their automation so that in turn we can describe how transparency shapes user agency during computational tasks and in digital environments.

Yet an awareness of transparency alone is not enough to explore the cultural significance of its self-effacing automation. It is equally important to understand how and why software designers arrived at the consensus that it should be the primary standard for computer interfaces. Although today's design standards were influenced by debates taking place in engineering research during the 1980s, their widespread implementation has occurred through a near total control of the market for personal computer OS software by Apple and Microsoft. Despite a rich body of work exploring the cultural work of computing from scholar such as such as Ian Bogost, Tom Boellstorff, Matthew Fuller, Alexander Galloway, Mark Hansen, Henry Jenkins, Jesper Juul, Nick Montfort, and Lisa Nakamura, there has little attention paid in the digital humanities to the personal computer itself. As Bruno Latour notes in *Reassembling the Social* (2005) , non-human objects function as tools for the solidification of social power: they are actors that "don't sleep" and can therefore form "associations that don't break down", at least not as readily as those among human actors, "allow[ing] power to last longer and expand further" (70). Although

Latour's charge that social theory ignores objects may not seem applicable to the study of digital media, transparency effectively removes many component objects from our view of media assemblages. Even within digital media scholarship, personal computers are treated like blackboxes, as if "they offer no information to the observer and will [appear to] have no visible effect on other agents. They remain silent and are no longer [acknowledged as] actors" in networked digital environments (79). This chapter contributes to a pressing need for a cultural history of personal computing. Before we set about pulling back the interfaces that hide complex systems from us, we must first interrogate the discourse surrounding transparency's emergence so that we can recognize the cultural assumptions about users embedded within those systems by their developers.

Accounting for transparency therefore requires a technical solution, both in terms of the critical theory used to describe its effects and the methodologies used to document its operations. Drawing on the concept of layered computing from theories of operating system design, Lev Manovich proposes in *The Language of New Media* (2001) that software consists of a "cultural" layer and a "computer" layer—or the software as visible on screen and the executed codes that comprise it (46).[3] Most projects undertaken in the 1990s and 2000s to study digital culture do so from the perspective of the cultural layer. According to Manovich, we can come to understand the computer layer in the way that it "transcodes" cultural objects, producing a "composite . . .computer culture—a blend of human and computer meanings" (46). The encoded structure of

[3] See *Structured Computer Organization* (2006, 5th Ed.) or *Modern Operating Systems* (2008, 3rd Ed.), both by Tanenbaum, for examples of the type of theory Manovich is drawing on from computer science.

new media allows us to bend and reimagine older media forms such as film and music as well as produce new hybrid forms through interactivity such as video games and electronic literature. Understanding how this composite culture is different from pre-digital culture is, according to Manovich, key to understanding these new technologies.

Yet transparency's push to efface computational operations suggests that there may be a disconnect between these two layers. Just as Bolter and Grusin's work implies that hypermediacy paradoxically contributes to an experience of the opposing logic of immediacy, so too does hypermediacy continually direct our attention away from Manovich's computational layer. Transparent design exploits software's inherent invisibility to represent computational operations selectively. Representing a computer system as a desktop where processes can co-exist, side by side in windows or arranged in neat rows of icons waiting for selection, hides the recursively nested file system structure that holds all of those processes in separate locations and manages data access between them. The desktop is essentially a functional misrepresentation of a computer system's structure. System designers and users thus hold different perspectives on computer systems. As users grow more accustomed to thinking of their machines in terms of functional misrepresentations, the computational layer and its coded algorithms slip out of our reach, becoming an exclusively technocratic domain. The idea that interfaces misrepresent computation does not mean that those studies of software's cultural layers are somehow made obsolete or incomplete; rather, an awareness of transparency can help us document and theorize how software's self-representation via broader cultural signifiers is re-shaping our understanding of race, gender, and sexuality. Research on digital cultural conducted from the users' perspective must continue, but we must also extend critical theory beneath the interface by resisting transparency and rendering visible underlying computational processes.

In this regard, I take a historical approach to reading source code, one that considers the specific contexts in which code is written, distributed, and executed, in order to explore the specific functions it carries out on behalf of its users. There are, however, a number of limitations and drawbacks to any methodology for the study of source code. Chun's charge that source code is often fetishized as logos, its limitations ignored so that it can be treated as the source of all truth and action in computational environments, is compelling (19-20). It is indeed impossible to "know" software, as she puts it, given the sheer size of the code base for most modern software applications and the number of other software libraries a given piece of code might depend on during execution, or the way drivers might translate software commands into hardware instructions. Source code can, at best, describe only a small fraction of what software might be doing at any given moment.[4] At the same time, however, a functional view of source code—one which understands it as instructions describing both what commands are available to users and how those commands are executed—can reveal cultural assumptions that aren't otherwise visible about the way software tacitly shapes user behavior. As Willard McCarty notes, the act of modeling something in code can itself be a critically self-reflective experience: a mode of "disciplined guesswork" that helps us to explore our tacit awareness of things (46-47). Developing software, testing it, and observing how it fails to meet expectations can expose latent assumptions about the processes being encoded. Importantly, McCarty's description of debugging his linguistic models suggests that reading code with a consideration of the data it interacts with also implies that a consideration of the structures visible in code can help us to

[4] See also Fuller's characterization of assemblages as always decomposable into other assemblages in *Media Ecologies: Materialist Energies in Art and Technoculture* (2007).

understand the cultural work of software. Software is written to be used, and while authorial intent remains a problem for any hermeneutic strategy, reading source code allows us to examine the assumptions latent in transparency's functional misrepresentations about who users are, what computers should be used for, and how software privileges or denies particular relationships among users.

While I am not the first to propose the critical analysis of source code, I contend that the problem of transparency is one that must be addressed, not by treating the hermeneutic study of source code as an end in itself, but rather as a means to investigate larger questions about its instrumental role in shaping structures of technological power. Although methodologies such as the "critical code studies" approach espoused by Mark C. Marino can help critics to develop a language to understand algorithms as aesthetic forms in terms other than simply elegance or efficiency, a functional approach to source code, coupled with an awareness of its extra-functional contexts, can tease out the moral, ethical, and political dimensions of design. As Richard Coyne observes in *Designing Information Technology in the Postmodern Age* (1995), information technology is often positioned as "beyond ethical," and I would add beyond criticism, because it is framed by a "rhetoric of needs" (77). The development and spread of new information technology often occurs under a banner of progressivism: there is a social or cultural problem in need of a solution which new technology can provide. Treating technology as a value-neutral resource that only assumes an ethical dimension through its application divorces design from social and cultural theory, allowing technological development to proceed "unimpeded by anything other than the constraints set within [the designer's] own technical problem-solving domain" (77). An important implication of Coyne's observation is that cultural studies of technology must attend to the ways in which cultural questions about technology are

often hidden by or reframed as requiring only technical solutions. Technology, in other words, also serves to functionally misrepresent sociocultural problems in ways that make it seem like an ideal solution. Yet methodologies which study code as an aesthetic form rather than placing its instrumental function within a cultural context risk replicating this problematic separation between the technical and the cultural. A critically instrumental reading of source code emphasizes that source code itself contains value judgments about who should use software, how it should be used, and for what purposes.

Guiding this contextualized, functional reading of code should be a critical awareness of software's history. Reading software through a rich intertextual archive can also help the digital humanities highlight the cultural importance of its primary texts. In her review of Matthew Kirschenbaum's *Mechanisms: New Media and the Forensic Imagination* (2008), Johanna Drucker controversially comments that the digital humanities has done a poor job of explaining the significance of new media criticism. She notes that digital media critics have focused primarily on "early and self-conscious works whose reflection on production is part of their textual condition." While I am hesitant to embrace Drucker's concern that software studies focuses too much on the "novel" because it devalues the aesthetics of software, I agree with her observation that the digital humanities has focused its critical attention on genres that have a circulation limited largely to academic circles, such as electronic literature and experimental software. Digital media studies must address software that doesn't have a readily accessible narrative or aesthetic component if it is to effect any change in the politics of software ecosystems. If the sort of born-digital texts she takes issue with are ones that lend themselves to primarily aesthetic investigations and close readings, then a critically instrumental methodology must look to texts which have a larger role in structuring models of computing—whether they

are video game engines that shaped an entire genre or the comparatively mundane, non-narrative software libraries that underlie digital content creation and distribution—and historicize their functionality.

At stake is not just a preference for one company and its particular model of computing over another. If we understand that design practices reify beliefs about who should use computers and for what purposes, then we must acknowledge that the authors of computer software are in many ways architects of social and cultural policy. Lawrence Lessig's oft quoted phrase, "code is law," is true in the sense that highly automated, transparent interfaces allow us to perform complex tasks, but only in authorized ways, while passively forbidding through omission those behaviors not described in source code. For expertly skilled users who know how to inspect application directories, navigate registries, and manipulate encrypted files, it may well be the case that no use behavior is forbidden; however, for the average user, transparent interfaces bar them from the domain of the technocrat. Now that broadband Internet connections are expected of most users by software developers, OS software, digital distribution platforms, and even individual applications serve not just to help users manage data on their machines but also to monitor and in some cases police the behavior of users. It is therefore vitally important for digital humanists to turn their attention to human-computer interaction (HCI) literature and understand not only how usability theory has been applied to develop new software but also how it is being interpreted to justify a model of computing in which users do not own the software they install and are permitted increasingly less control over their personal computers.

I. A Brief History of Transparency in Theory

HCI is a multi-disciplinary field situated within computer science that throughout its history has attracted researchers from the social sciences and the humanities. Cognitive

psychology has maintained a strong influence due in no small part to the work of Donald A. Norman, whose work from the 1980s and early 1990s is still taught as part of HCI courses and cited in its literature. Norman proposes in "Cognitive Artifacts" (1991) that information technologies don't necessarily change the way we think. Although Norman names no one who promotes this idea, he is responding implicitly to engineers like Alan Kay who cite Marshall McLuhan as a strong influence on their work.[5] Instead, Norman argues that tools like personal computers change the nature of the tasks we perform with them. From the cognitive perspective of the user, as opposed to say that of systems theory, creating a list does not extend our memory. Rather, it reduces the role that human memory plays in a given task. Instead of having to remember all steps, users must only remember to check the list after completing each step (20-21). Any perceived enhancement to users' cognitive ability is the result of reducing their cognitive burden. Although not as frequently cited as *The Psychology of Everyday Things* (1988), Norman's essay highlights a fundamental principle found throughout his body of work. [6] Information technologies can also increase our cognitive burden if designed poorly, and the engineer's goal should therefore be to fit technology into our lives in ways that makes cognition easier. The problem with software, he notes in "Why the Interface Doesn't Work" (1990) is that

> Interfaces get in the way. I don't want to focus my energies on an interface. I want

[5] On Kay's discussion of McLuhan and computing, see "Computer Software" in *Scientific American* 251 (1984) or "User Interface: A Personal View" in *The Art of Human-Computer Interface Design*, Ed. Brenda Laurel (1990).

[6] Norman's book was re-released in 2002 with a new title, *The Design of Everyday Things*, and an extended preface. All page numbers herein refer to the 2002 edition.

> to focus on the job. My tool should be just something that aids, something that does not get in the way, and above all, something that does not attract attention and energy to itself. When I use my computer, it is in order to get a job done: I don't want to think of myself as using a computer, I want to think of myself as doing a job (210).

If Apple is the company that established transparency as a practice, Norman's work establishes a theoretical basis for its continued application to design. Importantly, his work reflects HCI's role in the technocratic shaping of computational layers of software to produce cognitively "intuitive" interfaces that aid human productivity.

According to Coyne's account of HCI's theoretical orientations, HCI begins with what he describes as a conservative, "rationalist" tradition that gives way to the "pragmatic" one that came to define personal computing software in the early 1980s. The rationalist tradition, he notes, "promotes a particular view of the clientele or end users of technological systems," one in which "[t]heir participation in the process of inventing and designing computer systems is minimized" and in which only "experts have access to the theories and are best placed to deliver the appropriate designs" (28). Although HCI coalesced as a discipline in its own right in 1982 when the Association of Computing Machinery's Special Interest Group on Social and Behavioral Computing was renamed the Special Interest Group on Computer-Human Interaction, its rationalist assumptions can be traced back to the cybernetic theories of Norbert Weiner and research in early radar systems during the 1950s.[7] In particular, the Goals, Operators, Methods,

[7] For example, see Paul Fitts, "Human Engineering for Effective Air-Navigation and Traffic-Control System." Report. Washington, DC: National Research Council. Chun's discussion of the

Selections model (GOMS) proposed by Stuart K. Card, Thomas Moran, and Allen Newell shares with Weiner's cybernetics an assumption that the human mind is fundamentally an information processor structurally similar to a Turing machine. The GOMS model, at least initially, led HCI to strive for a quantifiable efficiency in goal-directed behavior. According to GOMS, software interfaces should be designed to minimize the number of actions required to perform tasks. The developer should therefore observe user behavior for the sole purpose of eliminating redundant or superfluous actions. The model GOMS does not address how users understand software, defining their relationship to software solely through the combinations of keystrokes they make with a Taylorist focus on timing task performance. GOMS, in short, privileges the perspective of the expert. The pragmatist orientation, according to Coyne, is more concerned with how "technology fits within the day-to-day practical activity of people," and sees as its mission the spread of computing power into as many areas of cultural practice as possible (31). Norman's work, as well as that of John M. Carroll, Alan Kay, Brenda Laurel, Ben Shneiderman, Lucy Suchman, Terry Winograd, and Fernando Flores, tries to describe the subjective experience of design decisions by users. But Coyne's interest in this transition is less in mapping the discourse surrounding it than in providing a stronger foundation for what he calls the "critical" and "radical" orientations to design. Both of these orientations are responses to the pragmatists and, Coyne argues, must be adopted by humanists and social scientists who study technology if we want models of design informed by critical theory.

Upon closer inspection, the writings of these computing pragmatists do object to the

---

SAGE defense system also highlights similarities between research in feedback system for military technology and early interface design theories.

rationalist tradition's inability to consider technology from the perspective of non-experts, but the solutions they provide are ones which subsume, rather than remove, rationalist features of design through more humanized representations of computer systems. The pragmatist orientation emerges through a rough consensus that variance among users makes it difficult to assess usability without considering the way cultural influences produce different subjective experiences of technology. Automation of task sequences coupled with an interface that visualizes feedback for users would marshal advances in computing power, placing new techniques for hypermediated graphics in the service of a perceived immediacy, changing the goal of design from efficiency to producing a form of interactivity that allowed users to manage their cognitive uncertainty. As interfaces became transparent, they reasoned, users would no longer be at the mercy of what Ted Nelson called a "computer priesthood" to dictate a model of use behavior to the public. Yet the call to make computing "invisible" and "unseen" is realized by exploiting the layered architecture of modern systems to produce computing systems wherein users become responsible only for manipulating the metaphors visible on screen. Those layers of algorithms below those metaphors would be handled by automated feedback systems, defined by developers and not directly accessible by users. The pragmatists, in short, saw functional misrepresentation of computational processes as the only way to guarantee that users, whether or not they had any prior training or familiarity with computer system, could readily develop a working understanding of software. Even in the "user-centered" approaches of the pragmatists, the unseen management of computer systems remains the domain of a computer priesthood, technocrats who approach the cultural problem of the increasingly complexity computation as one to address with more software.

The earliest research in HCI focused primarily on measuring software's usability by

employing a mechanistic model of humans as information processors. Not to be confused with behaviorism, cognitive psychology posits a universal, mechanical view of the mind: no thought or experience is direct but instead is the product of mental structures shared by all humans which interpret, recall, and store information. Ulrich Neisser, whose *Cognitive Psychology* (1967) served a foundation for the field, describes the study of cognition as the empirical observation of internal mechanisms that produce mental states, processes which collectively form behaviors but which should not be confused with more abstract psychological concepts like emotions, goals, or motivations that observers can only speculate about. Although Neisser comments in "The Imitation of Man By Machine" (1963) that computers are "oversimplified" metaphors for human cognition insofar as human thinking "serves not one but a multiplicity of motives at the same time" and "always takes in. . .a cumulative process of growth and development," he nonetheless invites the comparison in *Cognitive Psychology* (195). Much as Norbert Weiner proposed that humans and machines could be considered mathematically equivalent when functioning as part of the same information system, Neisser argues that a cognitive process is like a piece of software in that sense that both are "a series of instructions for dealing with symbols;" and even though software is "nothing but a flow of symbols it has reality enough to control the operations of very tangible machinery that executes very physical operations. A man who seeks the program of a computer is surely not doing anything self-contradictory!" (3). Neisser appears to reject cybernetics by noting failed attempts to apply Claude Shannon's information theory to quantify psychological processes; however, his description of humans and machine cognition is functionally equivalent to Norbert Weiner's mathematical description of human-machine systems. By treating both as information processors, software engineers could design computer input and output to complement users' cognitive structures. Importantly, Neisser's influence over

early HCI established the perspective of the expert as essential to user interface design, positioning the trained observer as more capable of understanding how users think about computers than users themselves.

First proposed in 1979 by Stuart K. Card, Thomas P. Morans, and Allen Newell, the GOMS model is representative of methodologies that incorporated empirical practices from cognitive psychology. In *The Psychology of Human-Computer Interaction* (1983), Card, Morans, and Newell recommend that designers write out user behavior in a form which more or less resembles pseudocode, a style of describing algorithms in natural language typically used to outline the structure of software before writing the particular operations in a programming language. Just as programmers must break down complex tasks into discrete algorithms, Card, Morans, and Newell assert that the user's mind similarly "segments the larger task [like] editing [a] manuscript into a sequence of small, discrete modifications, such as to delete a word or to insert a character," even if his or her awareness operates at a more abstract level (140). Briefly, GOMS assumes that users approach a computer with an abstract goal in mind—such as writing an essay, sending an e-mail, or searching for information—understand software as a finite set of operators which they can manipulate to perform their goal, know how to construct one or more methods combining those operators to achieve their goals, and then select the appropriate method based on the state they find the computer in at any given moment. By breaking tasks down into a series of small acts of observation, decision, and execution, GOMS allows researchers to redefine usability in terms that can be measured empirically, whether by the number of decisions, the number of physical movements, or the amount of time it takes to perform either. In this sense, GOMS understood the relationship between human and machine as largely informational. Software designers should not hide operations from users; rather, they must structure access to

those operations in ways that do not hinder users from finding and interpreting system information relevant to their goals.

Yet GOMS' practitioners, including Cards, Morans, and Newell, acknowledge that this method is less applicable to designing software with novices in mind. When users encounter a program state where they cannot determine which method will allow them to attain their goal, they must try combinations of available operators based on their memory of previously successful methods. In the process, users become "more skilled" by revising previously successful methods for use with new goals (363-372). GOMS' emphasis on software's internal consistency has remained popular, at least in experimentalist circles, because the observation of users performing familiar goals allows researchers to predict how they will respond to unfamiliar situations. GOMS assumes a goal-directed model of use in the sense that ideally users will know enough about a system's operators prior to sitting down at their machine to know which goals are possible, even if their system knowledge is incomplete or modified during use by the discovery of new operators and methods. While not as much a problem today because GUIs readily provide users with lists of operators, allowing users much greater leeway in assembling methods ad hoc, the model's reliance on users' prior knowledge of computers privileged advanced users during the dominance of CLI systems in the 1980s. Despite continued experimental interest in GOMS in ergonomics studies and the development of specialized computer software where efficiency and accuracy remains priorities, such as medical software, GOMS is no longer a primary heuristic in HCI. This diminished interest in GOMS reflects the personal computer industry's shift in 1984 towards a model of use that does not presuppose any specialized knowledge or experience.

The initial steps towards transparency are therefore ones which shifted away from the empirical perspective of the expert and towards models that tried to describe use from the

perspective of the user by acknowledging embodied practices. Ben Shneiderman's *Software Psychology: Human Factors in Computer and Information Systems* (1980), for example, discusses methods for applying observational techniques from cognitive psychology to a more holistic model of computing. Shneiderman proposed that systems should account for users with different degrees of technological literacy, using their machines for distinct purposes, and in different physical contexts. Each of these factors contributed to the ability of users both to define and to accomplish their goals, and each had to be approached differently by engineers. In terms of interface design, Shneiderman identified the memorization of terminal commands, the desire for closure when performing tasks, anxiety when unexpected states are encountered, a desire for control, and the speed and quality of software's response to commands as key problems that all designers must address (216-232). In this sense, Shneiderman's work recommends an attention to the qualitative variables influencing users' cognitive interaction with computer systems. Importantly, Shneiderman also observes that use practices are influenced by cultural attitudes towards computing. He expresses concern over a "gaping chasm between social commentators who perceive technology, particularly computer technology, as dreadfully harmful and computer science researchers who feel that computer technology can lead to a better way of life" (271). Shneiderman then closes his remarks by noting that the success of computers ultimately will be measured by the "willingness of each designer and implementer to taking responsibility" for the way his or her software will affect users' lives (280). By highlighting a disconnect between cultural and technical understandings of computers, Shneiderman's work attempts to position HCI as a discipline capable of mediating between the two perspectives.

The paradigm shift from a rationalist to a pragmatist orientation is marked by a concern that the spread of computing will fail to live up to its promise if users see it as a complicated

intrusion into their lives. Lucy Suchman introduced in *Plans and Situated Actions* (1985) what HCI engineers have since referred to as the "ethnographic approach" to design. Suchman's work incorporates anthropological and linguistic models that challenge the influence of cognitive psychology in HCI. Clifford Geertz's theories, for example, treat culture as consisting primarily of symbolic interactions, ones which require a "thick description" of the context of social interaction to understand. Cognitive descriptions of cultural practice, on the other hand, position themselves as objective by sticking to "thin descriptions" that observe decision making and behavioral rituals but avoid interpretation. But according to Geertz, cognitive anthropology leaves open the question of "clever simulations": meaning making becomes private in the sense that the social performance of various actors can be "logically equivalent" but understood by them in "substantively different" ways (Geertz 11). This problem of "mutual intelligibility" requires anthropologists not just to observe cultural practice but also to determine how actors continually construct meaning through a shared context. Observational methodologies like GOMS, in other words, ask only how systems could be redesigned to encourage ideal use behaviors and not why users failed or struggled to accomplish tasks. By observing workers in offices, Suchman concludes that while users are good at discovering methods during operation they may not understand why some fail and other succeed. Computers, she notes, are so complex that the "overall behavior of the computer is not describable, that is to say, with reference to any of the simple local events that it comprises"; instead, most acts of use occur only by comprehending abstractions representing "the behavior of a myriad of those events in combination" (15-16). Programmers should therefore approach the question of interface design not as a project of trying to design algorithms which serve as structural compliments to users' cognition but rather allow users to alter their use practices *in situ* in order to manage their

uncertainty about the system.

Suchman's conclusions implicitly challenge the technocratic management of cultural technology. In her closing remarks, Suchman suggests that "as long as machine actions are determined by stipulated conditions, machine interaction with the world, and with people in particular, will be limited to the intentions of designers and their ability to anticipate and constrain the user's actions" (189). Prior to her ethnographic model, usability problems were explained as the failure of designers to map users' cognitive responses to technology and to plan for all possible use behaviors. Until information technologies are designed to encourage users to develop their own use practices, rather than conform to those embedded in software, she suggests, engineers will control not only the technical production of computers but also their role in cultural production. David Golumbia writes more recently in *The Cultural Logic of Computation* (2009), for example, that "[c]omputers come with powerful belief systems that serve to obscure their real functions, even when we say we are acutely aware of the consequences of our technologies." (13). Golumbia's description of computation as a cultural logic links narratives of personal empowerment to bureaucratic applications of computation aimed at quantifying labor. Understanding computation purely in terms of efficiency encourages a worldview "wherein mathematical calculation can be made to stand for propositions that are themselves not mathematical, but must still conform to mathematical rules" (14). More politically charged than Suchman, Golumbia argues that we need to develop ways to discuss how computation reshapes the cultural contexts that software developers integrate it into because these technologies are continually presented to us as nothing other than benevolent. Understanding HCI primarily in terms of efficiency leaves "technocrats and capitalists to insist that we as citizens have no right or power to determine technologies change, adapt, and function

in society" (26). The rationalist orientation, in short, has not disappeared entirely. What Suchman implies and what Golumbia fears, in other words, is that that putting sole control in the hands of developers to define use practices will alter the way users think about the cultural contexts they use computers within, reshaping them to fit the algorithmic processes embedded in software.

Although Suchman's work sees human-computer interaction within social and cultural contexts, in practice designers have sought primarily technical solutions to the problems her work raises. Particularly, Shneiderman's 1983 essay "Direct Manipulation: A Step Beyond Programming Languages" has had a much stronger influence on commercial software design. When considered alongside Suchman, Shneiderman's essay highlights transparency's obfuscation of rationalism. According to Shneiderman, the same techniques of hypermediation to represent algorithmic processes in video games, computer-aided design and drafting tools, and spatial mapping tools could be applied generally to all forms of software. Video games are exceptionally well designed in terms of usability, he notes, because they can be understood without extensive training. Instead, the "general idea of the game can be gained by watching the . . .automatic demonstration that runs continuously on the screen, and the basic principles can be learned in a few minutes by watching a knowledgeable player" (61). Shneiderman explains that the practice of computing can be decomposed into acts of "syntactic knowledge" and "semantic knowledge" (65). Syntactic knowledge encompasses the commands users must memorize to complete a task as well as the precise syntax for directing and combining them. Although there may be some overlap or structural features common across most systems, Shneiderman notes that syntactic knowledge is largely "system dependent," requiring users to retrain themselves to some degree each time they use new software (65). By comparison, semantic knowledge refers to abstract tasks that are "natural. . . .and closer to innate human capabilities" (66). Presenting

computational processes visually in ways that establish spatial relationships between them, he argues, encourages users intuitively to model system structures and discover operations during use. The comparative lack of anxiety users experience when playing a game is due to a feeling of "direct manipulation" produced by immediate visual and auditory feedback to commands. When combined with tools that allow users to undo actions or revert system states readily, direct manipulation encourages users to explore systems during use rather than requiring them to learn about them in advance. Although Shneiderman's description of direct manipulation does not call for transparency's selective obfuscation, his framing of hypermediation suggests that the difficulties Suchman observed in users' attempts to explain syntactic structures could be solved with functional misrepresentations of algorithmic processes.

A key difference between Suchman's and Shneiderman's proposed solutions to efficiency based approaches to software design is that his leaves the relationship between designers and users largely unchanged. In both approaches, designers maintain primary control over the cultural work of computation by shaping the tools of cultural production; however, Suchman reframes design so that a programmer's technical expertise is necessarily incomplete. Instead of programmers relying on empirical models of cognition or quantifiable efficiency, they define usability qualitatively in conjunction with the cultural practices into which it will be incorporated. Shneiderman maintains that technical expertise is itself sufficient to account for variations among users and that the role of the designer is to develop tools which paper over the syntactic operations of computer systems—knowledge of which is either inconvenient, burdensome, or otherwise unnecessary for users. Direct manipulation, in this sense, is a strategy of creating increasingly indirect representations of computational processes which engage users on a pre-linguistic level, as "objects [which are] more 'natural' and closer to innate human

capabilities: action and visual skills [that] emerged well before language in human evolution" (66). Designers, in other words, can assume that they have a better understanding of users' subjectivity than users themselves. Whereas Suchman asked users to consciously reflect on how and why they chose specific use behaviors, direct manipulation appeals to scientific models of human consciousness to avoid such questions. Today, however, HCI studies often apply Suchman's insights in ways that ignore her points about the importance of understanding how users understand technology. Ethnography, as practiced in HCI, observes users in context of work they perform with computers but leaves in place the rationalist distinction between users and developers, which transparent design in turn reifies. This research focuses on how and why users respond to strategies of functional misrepresentation, not on how users understand the underlying systems.

One of the first studies to advance explicitly the idea that techniques of direct manipulation should be used to functionally misrepresent computational processes is Terry Winograd and Fernando Flores' *Understanding Computers and Cognition: A New Foundation for Design* (1986). Presented as a holistic intervention into HCI, the book begins by announcing that the important questions in software design are no longer purely technical and instead require a reflection on "human nature and human work" and a recognition "that in designing tools we are designing ways of being" (xi). By shifting HCI's perspective from considering "what [computers] do, not just how they operate," their aim is to consider how software transforms the nature of the tasks it models for users (4-7). Winograd and Flores challenge the rationalism of HCI using the philosophy of Martin Heidegger, arguing that "the practices in terms of which we render the world and our own lives intelligible cannot be made exhaustively explicit" (32). Like Suchman, they recognize that although "[w]e do at times engage in conscious reflection and

systematic thought. . .these are secondary to the pre-reflective experience of being thrown in a situation in which we are always already acting" (71). When we use tools, we actualize the "ready at hand" knowledge embedded with their design. Understanding a hammer does not involve knowing why, for example, a hammer is balanced in a particular way; rather, we understand its "hammerness" only in the act of driving a nail (37). Winograd and Flores see support for this relationship between humans and tools in Humberto Maturana and Francesco Varela's theory of autopoiesis. As self-organizing systems, human cognition is distributed in the sense that it is embodied in the structural couplings we form with our environment. Cognition takes place in our pre-conscious sensorium and biological responses to environmental changes as well as in the conscious decisions we make in our attempts to direct self-organization. For Winograd and Flores, Maturana and Varela's empirical efforts to describe a biological basis for cognition provide further support for the idea that users' primary engagement with tools is always pre-conscious. Any conscious engagement by users with mental models of technological functions is always already influenced by the systemic coupling formed between user and environment (70-73). Even though Winograd and Flores want to position their work as describing a cultural approach to computing, they nonetheless reproduce the assumption Suchman saw as problematic: that users' conscious understanding of technological structures is less valuable in design than an expert's understanding of scientific models of cognition.

While Shneiderman implies a need for functional misrepresentation, Winograd the Flores argue explicitly for an "ontological" design that strives for a "transparency of interaction." Winograd and Flores propose, in terms similar to Shneiderman's concept of direct manipulation, that designing software should not be much different from designing a car. Just as while driving a car "[y]ou do not think 'How far should I turn the steering wheel to go around that curve?'";

users should not be required to think about how to operate their computer in order to accomplish tasks with it (164). Drivers do not need to know how an engine works to benefit from its functions, and so too users shouldn't need to understand a system's computational layer to participate in the cultural layer. Unlike an engineers who design cars, however, a programmer has much greater control over technological form: they quite literally design the structures which "create the world in which the user operates" (165). Creating transparent technologies that allow for direct manipulation therefore requires programmers to design "ontologically clean" environments that afford distinct frames of action. In Shneiderman's terms, Winograd and Flores call for designers to maintain a strict separation of the syntactic elements of the system from semantic representations of tasks whenever possible. Although word processing programs once required users to manage modes—view, insert, overwrite, or command—there was no physical limitation preventing designers from producing software that resembles today's office software, which just lets users write.[8]

Separating users' experience of the cultural layer from a system's computational operations does not remove the former's dependence on the latter. Winograd and Flores' description of "ontologically pure" domains relies on interfaces that automate syntactic tasks to provide an illusion of seamless, direct manipulation. Their discussion of mail servers highlights the idea that this separation also establishes a model of use where a large population of users' computational practices remain dependent on informational structures shaped by a much smaller group of programmers: "The user operates in a domain constituted of people and message sent

[8] For an example of an older style word processor, see vim, itself an updated version of vi: http://vimdoc.sourceforge.net/htmldoc/intro.html

among them. This domain includes actions (such as sending a message and examining mail) that in turn generate possible breakdowns (such as the inability to send a message). Mailbox servers, although they may be a critical part of the implementation, are an intrusion from another domain—*one that is the province of system designers and engineers* [emphasis added]" (165). Winograd and Flores' transparency of interaction is therefore predicated upon maintaining a separation between domains appropriate for users and those best left to technical expertise. As Manovich notes, HCI's foremost achievement has been to change computing so that "we are no longer interfacing to a computer but to culture encoded in digital form" (69-70). The term "cultural interface," he argues, thus becomes more appropriate than "computer interface" because users understand themselves as interacting primarily with "cultural data" rather than the computational forms which transcode it (70). Manovich's description of interfaces is part of an effort to reinterpret them in terms familiar to existing hermeneutic practice them by highlighting the way they participate in a historically rich media ecology. At the same time, Manovich's idea that the computational layer is continually reproducing the cultural layer through a process of transcoding reminds us that an ontological separation between the two is impossible. Transparent design, in this sense, enacts an epistemological separation. It is instead only a functional fantasy that users accept: a sublimation of computation that paradoxically makes computing seem more accessible.

Models of usability based on misrepresenting or selectively representing computation thus encourage a specific kind of technological literacy, one which is limited to manipulating the cultural interface but which yields very little information about computation itself. In reviewing the influence of technology on Heidegger's mediation on Being, Coyne describes a "darkening of the essence of truth" in the technological age as technology enframes the world according to

an instrumental logic (67-68). Heidegger's phenomenology understands technology as both instrumental and cultural. In "The Question Concerning Technology," he describes technology as a way of revealing but notes that what it uncovers about our world is dependent on our relationship to technology. According to Heidegger, technology has become inseparable from culture, and our turn to science to understand Being has made our search "dependent upon technological apparatus and upon progress in the building of apparatus" (14). The revealing that technology offers is therefore made available only in terms of itself, a challenging of old ways of knowing "which puts to nature the unreasonable demand that it supply energy that can be extracted and stored as such" (14). As it reveals facts about the natural world, it also conceals older ways of knowing truths. Although it would be a stretch to position computational processes as part of the natural world, this logic of concealing through the act of revealing is evident in the construction of transparent interfaces. The sublimation of computing power thus encourages users to cultivate a computational subjectivity that itself is dissociated from knowledge of computation. Even though, as Sherry Turkle's work suggests, we do form deeply personal symbolic connections to information technology, the use of software is increasingly a public act.[9] Not only, as Manovich argues, does computation now transcode the production of most other media forms, but the ubiquity of networked applications means that no single machine carries out computation in isolation. In this sense, all digital media produced by transparent interfaces contributes to a discourse about computing that seeks to conceal its technological nature behind a rhetoric of task-completion that emphasizes the role software enhances human information production but not necessarily human creativity.

[9] See *The Second Self* (1984) and *Life On the Screen* (1995).

Following Winograd and Flores, Norman re-establishes in HCI the expert's ability to model users' subjective interaction with technology by describing transparency's functional misrepresentations using scientific models of cognition. Like Winograd and Flores, Norman argues in *The Psychology of Everyday Things* that software interfaces "can be anything the designer wishes to them to be," giving programmers the option of shaping their software to "tak[e] on the appearance of the task" at hand and letting complex computational models "disappear behind a façade" of simplicity (183). Central to most of Norman's writings is the idea that designers should address what he terms the "gulf of execution" and the "gulf of understanding." Norman defines the gulf of execution as the "difference between the intentions [of users] and the allowable actions" of technological systems (51). Similarly, the gulf of evaluation "reflects the amount of effort that the person must exert to interpret the physical state of the system and to determine how well [their] expectations and intentions have been met" by the system (51). Drawing on his background in cognitive psychology, Norman notes that people always form mental models of how technology works during use, forming them ad hoc in "situations that are not remembered (or never before encountered);" it's therefore important for designers to "provide users with appropriate models" through interfaces which suggest what they consider to be correct use practices because "people are likely to make up inappropriate ones" without comparable experience or formal training (70). According to Norman, inaccuracies that prevent effective application of mental models to use practices are a combination of these two gulfs. To address these gulfs, Norman develops J.J. Gibson's concept of environmental affordances by adding a perceptual dimension: affordances are not just possibilities for action that exist independently of subjects but rather features of technology in which users recognize specific actions and which convey a sense of how those actions fit into larger task-flows (9).

Norman thus appears initially to share Suchman's conclusion that good design is one that takes into account users' subjectivity and that usability as a theoretical problem is one best understood as a symbolic interaction.

While Norman's emphasis on users' subjectivity suggests an approach similar to Suchman's, his methodology relies on a scientific definition of a universal subjectivity rather than a substantive engagement with users' symbolic practices. In "Direct Manipulation Interfaces," Edwin L. Hutchins, James D. Hollan, and Norman argue that usability is a question of cognitive distance. Shneiderman's concept of direct manipulation is thus best understood as "the commitment of fewer cognitive resources. Or, put the other way around, the need to commit additional cognitive resources in the use of an interface leads to the feeling of indirectness" (317). Norman comments in *The Psychology of Everyday Things* that constraints are more important than affordances because they reduce the number of decisions required to use a given piece of technology (62). Constraints are some feature of a technology, whether physical, semantic, cultural or logical, that forbid "incorrect" use actions (81-86). GUIs in this sense must be selective in their representation of computational processes. According to Hutchins et al., the only drawback to transparency is that some tasks which "do not easily decompose into the terms of the [high-level semantic image of the system] may be difficult or impossible to represent" (325). Designers therefore have a responsibility to produce an interface that acts "as an intermediary to a hidden world" of complex informational structures but which also "denies the user direct engagement with the objects of interest. Instead, the user is in direct contact with [symbolic] structures, structures that can be interpreted as referring to the objects of interest, but that are not those objects themselves" (319). Norman's work in this sense marks a return to the thin description of human-computer interaction. Unlike Suchman, he is not concerned with

understanding particular symbolic interactions between users and software; instead, he assumes that the correct application of technological principles can produce among users mental models that are functionally equivalent regardless of how accurately they understand the computational structures with which they interact.

At least in personal and business software, Norman's description of transparency largely remains one requirement for a given piece of software to be considered successful. Norman's theories aligned so closely with Apple's vision of the future of computing (see Chapter 2), in fact, that he began working as a consultant for the company in 1988 and eventually left his position at the University of California at San Diego in 1993 to work full-time for the company as a Vice President and "User Experience Architect." Apple's *Macintosh Human Interface Guidelines* (1992) uses terms similar to those found in HCI's academic literature, including the idea that software should be designed to "take advantage of people's knowledge of the world" and allow for a feeling of "direct manipulation" (4-13). Like Norman, Apple cautions designers to consider that software "often introduce[s] a new level of complexity for people" and that their projects should therefore strive to "provide a computer environment that is understandable, familiar, and predictable" (11). Although they do not cite Norman directly, Apple notes that their "general guideline that simple design is good design applies directly to the discipline of human-computer interaction" (35). Furthermore, Apple recommends that programmers design their software specifically so that they have very little reference to computational structures. Their "80% Solution" shares with Norman the assumption that well designed technology is one which simplifies users' perceptions of technology by providing them with feedback without revealing the details of its algorithms. Apple proposes to programmers: if "you design for the 20 percent of your target audience who are power users, your design will not be usable by the majority of your

users. *Even though those 20 percent are likely to have good ideas and probably think a lot like you, the majority of people may not be like the 20 percent who are elite users* [emphasis added]" (35). Although HCI theorists proposed transparency as way to make computers less intimidating, the industrial application of it that begins in the 1980s serves to monopolize direct access to computational structures. Programmers may use transparent design to put their software in the hands of more people, but it also serves a key secondary purpose of closing off the technical details of computer systems to everyone, regardless of their skills or knowledge.

II. Transparency and Public Computing

I do not wish to suggest that the solution to transparency's problematic obfuscation is for computer systems to be stripped of all technical automation or hypermediation in order to restore an idyllic direct relationship between human and machine. As Chun's study of the earliest days of computing indicates, any narrative which tells of a time when users did have direct access to computational processes is itself a myth. The practice of computing has always depended on an intermediary, whether it is a human manually making circuit connections or an operating system that controls hardware operations. According to Chun, contemporary computing concepts like "programming became programming and software became software when the command structure shifted from commanding a 'girl' to commanding a machine" (29). Even those women who acted as computers by manipulating circuitry never had direct access to computing power because they operated machines without understanding the processes they managed. Their role was simply to say "Yes, sir" and carry out the commands of engineers (30). Like these early women computers, the work of today's transparent software operating systems is removed from mental models of computing. Although we see the metaphors and environments they produce on screen, we rarely acknowledge the full extent of their agency as objects. We assume our

transparent interfaces stand at the ready, executing our command upon request.

However, there is a significant structural difference between transparency as discussed here and Chun's description of the earliest examples of human-computer interaction. In the system described by Chun, user and programmer were the same person. The human "computer" serves as the user's agent, and users instructed their agents directly. But in today's computer systems operating systems, user and programmer are not the same person. Software acts as an extension of the programmer's will—or the collective will of an team, an intersection of a corporation's business, development, and marketing teams—serving as the user's agent only according to the instructions written in advance by its designers. Although HCI literature continually promises that transparency will produce software that invisibly carries out users' commands, allowing them to focus on high-level tasks, it is more accurate to say that users select abstract representations of commands produced by a system's designers. Conceptually, users understand themselves to be opening files or surfing the Internet, but computationally they are initiating chains of commands that access data encoded in binary notation in specific physical locations, parsed into meaningful information from fixed sequences of bits, and further interpreted through intermediating libraries before being displayed on screen in human readable language or images. So long as the gulfs of execution and expectation are bridged by functional misrepresentations, the computational layer is beyond their reach and thus their concern. Users can confront computation only when transparency breaks down. So long as the subjective experience of transparency is maintained, designers are free to insert any number of mediating processes into the command chains that occur beneath the interface's cultural layer. The same principles of transparent design that make for "user friendly" software also make possible the keyloggers that blackhat hackers use to steal passwords and credit card numbers, the spyware

that reports on our Internet usage to advertisement agencies or the NSA, the digital rights management systems that police our media consumption, or the viruses that can corrupt our personal computers to the point where only a complete software wipe is the only hope for restoring them.

The problem of transparency does not lie in the separation of the computational and cultural layers into distinct knowledge domains, but in the assumptions that users should be forbidden from accessing computation and that interfaces must enforce this policy. Computers can no longer be considered "personal" in the sense that we access them in private contexts, if indeed any medium ever could. Today's computer systems, whether they take the form of smart phones, tablets, laptops, gaming consoles, or desktops, are designed so that the computational layer is in constant communication with information networks. As political subjects, users are now part of a system that resembles the public in the early twentieth century following the rise of the mass media that Walter Lippmann described in *Public Opinion* (1922). Just as Lippmann describes a conceptual distance between what people experience in their everyday lives and the national or international events reported to them, so transparent design instantiates a conceptual distance between the interface on screen and the processes it purports to represent. Transparent design shares much in common with Lippmann's solution to the problem of democracy in the age of mass media: the subjective experience of directness created through an engagement with hypermediation produces a model of use wherein computation can be directly managed only by the decisions—embedded in code—of a technocratic elite who believe they will always understand the nature of computational processes far better than the public. Unlike in Lippmann's view of the mass media, however, the role of the technocrat isn't to apply scientific knowledge to influence public opinion and manage public policy. As Lessig suggests, the writing

of code is always already the writing of law—or perhaps something more, as code's constraints forbid action pre-emptively. As Chapter 2 explores, the conceptual distance of transparency has placed designers in a role that allows them to shape our imagination of the future of technology. Transparency is in this sense not just a technical constraint for users but also a cultural one. With each generation of computer software comes the expectation of greater transparency, and companies like Apple and Microsoft represent technical advances in ways which increase this conceptual distance so that the future of personal computing has little to do with computation itself.

Transparent design also allows the producers of information technology to shape our cultural attitudes towards those operations occurring in the computational layer that we cannot see but nonetheless may feel the effects of. In recent years, the piracy of digitized media as well as personal computer software has become a hotly debated public concern. Media authors frame piracy as a pressing cultural concern, discussing access in terms of moral and ethical principles and arguing that users should understand access to digital media in the same terms they use to access physical media. Corporate copyright holders frame piracy as a pressing economic issue and argue that each illegal access of digital media represents a quantifiable loss of revenue. Software engineers, however, have addressed the problem primarily in technological terms: exploiting transparency in order to enforce copyright on a level that was not possible with physical media.[10] Technologies like digital rights management software realize the possibility of exploiting transparency to control cultural production. These same strategies also have been

[10] For an in-depth consideration of how software enforces copyright, see Lessig's *Code and Other Laws of Cyberspace: Version 2.0* (2006).

applied recently to efforts by Apple, Microsoft, and Google to control user choice in software. Popularly referred to as the "walled garden" approach to computing, package manage systems require users to access new software primarily though an "app store." On smart phones and mobile operating systems, users are by default limited to those applications which these companies have permitted to be included in the store. Although Apple, Microsoft, and Google state these models enhance system security by ensuring that only software from "trusted" developers can be installed, it also affords them substantial control over software ecosystems.

Finally, the Free and Open Source Software movements suggest that hypermediated, cognitively simpler interfaces need not also employ transparency's strict separation between the computational and cultural layers. Linux and other UNIX-like systems have highly modular structures that permit personal computer users to access, customize, or remove most system components. Users can engage in hypermediation's celebration of computation, enjoying GUIs that provide a comparable feeling of direct manipulation to commercial OS software with the difference that they do not have to accept any aspect of the fantasy of use presented to them. Engineers and programming enthusiasts contributing to the Linux software ecosystem thus avoid the blackboxing of commercial design and transparency's computation-less computing. As I argue in Chapter 4, culturally aware, instrumental readings of source code have the potential to benefit all users, not just those who are themselves experts in programming, because the free and open source models of design expose the political dimensions computational processes for public consideration. The Linux software community uses a diverse approach to technical problems, often designing particular distributions of the Linux operation system around specific political principles.

Chapter 2 – User-Friendly? Transparency and Personal Computing

"What we are trying to do is reach the point where the operating system is totally transparent. When you use a Lisa or a Macintosh, there is no such thing as an operating system. You never interact with it; you don't know about it."

–Steve Jobs, 1984 interview with *Personal Computing*

"People are wrong about technology. They talk about it in terms that are utterly, ideologically incorrect, as if it were a shiny box that beeps and gives you candy."

–Bruce Sterling, from an interview in *Across the Wounded Galaxies*

Following the introduction of the Apple Macintosh in 1984, the computer industry worked to change the popular image of computers from intimidating machines with complexities that were decipherable only to a select group of technocrats—the "IBM model"—into humanized appliances designed "for the rest of us." The conflict in personal computing between IBM and Apple that began with the release of the IBM PC in 1981 represents more than just two companies fighting over a share of the market because their products embodied two different philosophies of computing. While both invoked the idea that their machines were "easy" and "user-friendly," IBM's PC established a standard for a general purpose model of personal computing that strived to make technical information as well documented as possible. Apple, on the other hand, wanted to make personal computers designed for specific purposes, streamlining interfaces with intuitive visual metaphors so users could "see through" the machine directly to

the data they were manipulating. Paradoxically, this push towards what was called "transparency" by spokesmen like Steve Jobs involved the appearance of simplification by obfuscating and automating complex operations rather than making them more accessible to users. Jobs promised that in the future users would not need to notice their computers at all. Like Jobs, many in the industry would come to describe themselves as not just designing a new machine but of finally realizing the promise of the counter-cultural computing revolution that began in 1974: a new period in which the transparent interfaces of personal computers would free users from the burden of computing and make possible new forms of expression.[1]

The belief that future development of computer technology would be directed towards an ideal of transparent design was not unique to Jobs and other stewards of early personal computer systems. As Howard Rheingold's history of virtual reality indicates, William Gibson's vision of a transparent cyberspace in his Sprawl Trilogy influenced a wide array of commercial researchers developing new computer graphics applications. Gibson's cyberspace is in many ways the epitome of transparent design as users see through hardware, experiencing a direct connection to cyberspace. John Walker, who founded Autodesk and designed the widely used AutoCAD software, not only acknowledges Gibson as an influence on his software but in 1988 released an internal memo that called for an "Autodesk Cyberpunk Initiative" to find ways to integrate virtual reality's experience of directly manipulating data into his company's computer-

[1] Although most general histories of computing will cover this period, see Fred Turner's *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism* (2006) for a closer examination of the revolutionary discourse surrounding personal computing.

aided design and drafting software (183-185).[2] Scott Bukatman suggests that cyberpunk, and Gibson's work in particular, served to provide Americans with a vocabulary "to redefine the imperceptible (and therefore absent to consciousness) realms of the electronic era in terms of the physically and perceptually familiar" (117). Yet Bukatman could very well be describing Apple's effort to do the same thing in producing the Macintosh's desktop-based graphical user interface. Rheingold's own intermixing of Gibson into his discussion of the development of transparent computer graphics technologies suggests that cyberpunk influenced in complex ways the discourse about the role of personal computing.

Yet unlike industry insiders and other spokesmen, Gibson and fellow cyberpunk Bruce Sterling could not actually see through the obfuscation of transparency. Instead, they helped us to imagine what would be gained or lost in personal computing's turn towards transparency and what a computer culture based on hiding complexity might look like. Critics writing on the advent of cyberpunk literature and film, and particularly Gibson's *Neuromancer* (1984), recognize that cyberpunk strived to make visible "unseen forces" at work within a rapidly growing information economy. [3] I contend that these unseen forces critics refer to represent the

---

[2] Rheingold even notes that Walker was sued by Gibson's attorney for attempting to trademark the word "cyberspace" to promote his company's software.

[3] See David Porush's "Cybernetic Fiction and Postmodern Science" (1989), Terence Whalen's "The Future of a Commodity: Notes Toward a Critique of Cyberpunk and the Information Age" (1992), Scott Bukatman *Terminal Identity: The Virtual Subject in Postmodern Science Fiction* (1993), Thomas Bredehoft's "The Gibson Continuum: Cyberspace and Gibson's Mervyn Kihn Stories" (1995), Istvan Csicsery-Ronay, Jr.'s "Antimancer: Cybernetics and Art in Gibson's

design decisions and other work of developers obfuscated by the principles of transparent design. Sterling asserts in his Preface to *Mirrorshades* (1986) that "the techniques of classical 'hard SF' extrapolation, technological literacy" are, in a society that is quickly assimilating computer technology, "not just literary tools but an aid to daily life." Ideally, he argues, science fiction should pay careful attention to the precise mechanisms at work in these new technologies in order to tease out the role that hardware and software developers are playing in our culture. Yet as J. David Bolter and Richard Grusin observe, the popularization of graphical user interfaces like Apple's Macintosh represent an attempt by designers "to remove the traces of their presence in order to give the program the greatest possible autonomy" (72). Transparency thus poses a critical problem: as a philosophy of design it acts as a horizon of knowledge by passively denying users awareness of a system's mechanisms. It is not a question of searching for information and not finding it; rather, transparency strives to keep users from even considering what is kept invisible to them, accepting instead a proliferation of simplified representations. In acting as an aid to daily life, Gibson's Sprawl Trilogy helps us to consider the consequences of a society divided by gaps in technological knowledge. Cyberspace, in this context, serves to make visible not technical knowledge but rather the restrictions transparency imposes on our ability to access and interpret computational operations.

Conversations taking place in popular periodicals, computer magazines, and technical

---

'Count Zero'" (1995), Timo Silvonen's "Cyborgs and Generic Oxymorons: The Body and Technology in William Gibson's Cyberspace Trilogy" (1996), and E. L. McCallum "Mapping the Real in Cyberfiction" (2000) for examples of this critical trend. My phrasing is drawn from Bukatman's book.

journals note that by 1981 the sudden growth of the personal computer industry had created a chaotic marketplace that both punished and turned away potential new users. Since the release of the Altair 8800 in 1974, personal computing had been surrounded by a utopian narrative emphasizing the role of West Coast hobbyists in the Homebrew Computer Club. [4] Dovetailing with the early descriptions of the Internet as a platform for radical equality—no doubt in large part due to Stewart Brand's and Ted Nelson's writings—this account describes personal computers as the product of young, daring innovators who saw that computers could be used as creative tools, enhancing the lives of their users, rather than as simply sophisticated data processors—provided developers could make them accessible by most consumers.[5] Despite this narrative, there was a rough consensus that the benefits of these new tools had yet to reach the general public. Most commentators agreed that problems of accessibility needed to be addressed in order to sustain the industry. But as Charles Rubin explains in a 1984 essay for *Personal Computing*, there was a disagreement regarding how to make computing easier for users: "Should we be working toward software that facilitates the analytical process, so it becomes an

---

[4] The general history of personal computing in this chapter, unless otherwise noted, is a synthesis of Paul Ceruzzi's *A History of Modern Computing* (2nd Ed., 2003), Paul Freiberger and Michael Swaine's *A Fire in the Valley: The Making of the Personal Computer* (2000), and Martin Campbell-Kelly and William Aspray's *Computer: A History of the Information Machine* (2004).

[5] Specifically, Brand's article in *Rolling Stone*, "Spacewar: Fanatic Life and Symbolic Death Among the Computer Bums" (1972) and Nelson's *Computer Lib: You Can and Must Understand Computers Now* (1974). Larry McCafferty's *Storming the Reality Studio* (1991) also features a number of essays from countercultural figures promoting computers.

integral part of the decision-making process, or should the direction be toward automating that process so the user hardly thinks about it, concentrating instead on reaching the goal he is seeking?" (89). The two possibilities Rubin sees in the market represent distinct assumptions about the proper construction of computers. As a highly refined, well documented, and essentially open architecture, IBM's PC standard represents user-friendliness as a practice of exposing the technical specifications of a machine's design and letting third parties and consumers work together to find their own ways to integrate personal computing into their intellectual labor. Apple's designers, on the other hand, wanted to develop personal computers that would free users from the cognitive burden of computation by limiting them to the specific intellectual activities suggested by the visual icons of its graphical interface.

Just as the emphasis on facticity in the sciences obscures the cultural contributions made by scientific "failures," so too does the success of Apple's transparent design, the norm for commercial hardware and software, makes it difficult to see and understand the value in other models of computing. Even though Apple struggled financially for several years following the Macintosh's release, its philosophy of design changed the market, moving Microsoft to abandon many aspects of the general purpose IBM model it had supported with its DOS operating system in order to copy the automation of Macintosh's interface with Windows during the 1990s. Similar to Alexander Galloway's study of "seamlessness" in Internet software infrastructures, this project proposes that the eventual widespread adoption of transparent design was the result of its association with a countercultural narrative that rejected, at least on its surface, all forms of technocracy.[6] Nonetheless, the discursive power of transparency extends not just to removing the

[6] See Galloway, *Protocol: How Control Exists After Decentralization* (2004).

work of computing from discussions of computers but also from the history of computing. The emphasis which many scholars place on the first demonstration of graphical user interfaces by Douglas Engelbart at Xerox PARC in 1966 shows just how important transparency was to shaping cultural attitudes about computers. Transparency has, in a sense, rewritten the history of computing so that it has always been the goal of design. Wendy Chun, for example, invokes the idea of transparency when she notes that Engelbart's demonstration reveals that the function of software has been the production of neoliberal subjects through a process which grants users a feeling of power over information while at the same time "obfuscat[ing] the machine and the process of execution, making software the end all and be all of computation" (19). Historical narratives that privilege Engelbart's demonstration, and the way the Macintosh commercially realized its innovations, obscure the ways in which the de facto standardization of IBM's PC helped to secure personal computing's place in American society by stabilizing a chaotic hardware market and opening up a space for third-party hardware and software designers.[7]

Despite its importance in laying the foundation for phenomena that have fascinated cultural studies of digital media—Internet studies, cyberculture studies, video game studies, and new media studies—the personal computer itself has been largely overlooked. Bolter and Grusin's notion that digital media are always negotiating between the complementary desires for hypermediacy and immediacy suggests that transparent design is a negotiation between general purpose and specific models of computing. All personal computers, even those that are tailored towards specific purposes, are built upon the same core platform of general purpose processors.

---

[7] Bill Gates describes the importance of the standard for software companies in "A Trend Towards Softness" (*Creative Computing*, Nov 1984). On hardware, see also Ceruzzi, Chapter 8.

A sense of transparency is therefore produced by constraining the range of possibilities afforded by general purpose systems in order to direct computing power towards a set of direct manipulation experiences; or, to use Bolter and Grusin's terms, transparency results from narrowing the focus hypermediacy and using its expressive power to produce immediacy. Designing a transparent interface thus entails the rejection of all other models of use as incorrect, unproductive, or unauthorized. By studying the history and construction of personal computer hardware and software, we can not only learn how their design influences the larger scale digital projects they support but also how the knowledge of computing they afford us encourages a specific type of public discourse, one which is limited to a concern for what computers appear to do and not how they work nor how they are produced.

Placing the work of Gibson and Sterling within the context of these debates over models of personal computing not only helps to elucidate the aspects of transparent design that industry figures were unwilling or unable to consider but also helps to return to cyberpunk the critical edge that many recent literary critics have called into question. Sterling's work implies, for instance, that science fiction encourages us to imagine the "fossil record" of technology. Gibson's treatment of computers in The Sprawl Trilogy initially seems to take transparent design as a widely accepted and even desirable part of the future of computing, as indeed the use of these machines by Henry Case, Bobby "Count Zero" Newmark, Angela Mitchell, and Kumiko is so transparent as to be hardly recognizable when compared to interfaces comprised of windows, keyboards, mice, touchpads, and monitors. Imagining a future of transparent technology through science fiction exposes a potential fossil record or line of innovation that helps us consider the long-term cultural consequences of transparency. Because transparency forbids knowledge of technical specifics, a culture shaped by its principles leaves no room to discuss technological

innovation in terms other than those abstract, high-level experiences it permits. Transparency, in this regard, perpetuates itself by continually producing in users a desire for more transparency. Cyberpunk fiction's iconic trope of interfaces both haptic and neurological portray complete transparency through advancements in user-friendly design as inevitable. But upon closer inspection, many of the genre's troubling encounters with technology occur when characters face the limitations that transparent design practices place on their knowledge of and agency within cyberspace.

I. The IBM PC and the Apple Macintosh

An anecdote found in Andy Hertzfeld's Folklore.com, a web project collecting personal stories about the development of the original Macintosh, describes how Steve Jobs proposed near the Macintosh's completion in 1982 that each member of the design team have his or her signature molded into the inside of the machine's case. Hertzfeld explains that "[s]ince the Macintosh team were artists, it was only appropriate that we sign our work." What was intended to become a reminder of the craftsmanship behind the ultra-modern, mass produced exterior serves as a symbol for the divide between designers and users that the Macintosh solidified. As many reviews noted in the 1980s, the Macintosh was unusual in that it was the only high-priced personal computer to be hermetically sealed, requiring a special set of tools available only to officially licensed Apple technicians to open. The signatures of Hertzfeld, Jobs, and others would thus never be seen by the average user and so served to claim the interior as an exclusive space remote from the humanistic and creative rhetoric of the Macintosh's advertisement campaign. Appropriately, Hertzfeld's anecdote ends by noting that "over time, names gradually began to disappear for practical reasons, as Apple changed the case to make it easier to manufacture. Some details were changed even before first ship, partially obscuring some of the signatures.

Each time the case was revised, more names were left off, as dictated by the nature of the revision."[8] As the Macintosh was revised to be more and more transparent, the sophisticated engineering of the machine relegated creativity to the exterior, leaving the interior a tightly controlled and highly automated machine space.

The conflict between IBM and Apple that began with the former's entry into the personal computer market in 1981 with its 5150 model, the "Personal Computer" (PC), and reaches a conclusion in 1984 with the release of the ladder's Macintosh illustrates a movement away from a design philosophy of open, generalized architectures and towards one of automation and control. Talk of a personal computer revolution began in 1974 with the development of general purpose processors: chips with only the most basic of operations hardwired into them that would require an extensive, modular apparatus of hardware and software build around them to perform even the simplest computations. When asked by the now defunct Busicom in 1968 to design and manufacture a special purpose circuit, Intel—whose business was primarily in logic chips—instead decided to develop a general purpose microprocessor that could be reused for other contracts by connecting it into a network of attending chips, such as ROMs, that had specific computational functions burned into them (Ceruzzi 218-219). While the Intel 4004 had limited commercial success, its descendants, the 8008 and 8080, drew the attention of hobbyists like Steve Jobs and Steve Wozniack who started using them to build their own "personal" machines. These first machines could do very little. When it was initially produced in 1974 by a small company named Micro Instrumentation & Telemetry Systems, Inc. (MITS), the Altair 8800 was little more than a direct interface for the Intel 8080 processor: an aluminum case with of a series

[8] http://folklore.org/StoryView.py?project=Macintosh&story=Signing_Party.txt

of eight lights and corresponding switches on the front panel. Operation involved changing the processor's eight bits to 0 or 1 using the switches before manually selecting to store the instruction in memory and then to advance the processor to the next cycle. The Homebrew Computer Club and similar hobbyist groups formed to fill in the void left by a lack of documentation and functionality, with members modifying their Altair 8800s so that the machine could do more than simply flash the lights on its front panel using parts manufactured by MITS or by even smaller third-party companies.

Although the Altair 8800 a became site for the creative exploration of computing and served as a sign that the personal computer revolution really was freeing computing from the confines of corporate and academic mainframes, at the beginning of the 1980s it was unclear how the general public could participate without the extensive technical knowledge that hobbyists had developed through years of tinkering. Ad campaigns from both IBM and Apple presented two competing visions for the future of a "user-friendly" personal computer. Both companies responded to the chaotic marketplace for personal computers in the first half of the 1980s by presenting their machines as "easy" to use. IBM produced a well-documented, expandable architecture that would allow users who took the time to read its manuals to avoid making costly compatibility mistakes when they purchased hardware and software. Although the machine which made Apple a household name, the Apple II, was very much a product of the general purpose philosophy of computing, Jobs' vision of the Macintosh questioned the benefits of a model that relied on users possessing a sophisticated technological literacy. In this sense, the Apple Macintosh and IBM PC represent two different definitions of "user-friendly": the former a foundational example of the belief that technical knowledge is a burden and the latter a style that encouraged users to study deeply and come to master their equipment. Whereas the IBM PC

established a standard for general purpose machines that, by default, did very little automatically but allowed users great freedom to pick and choose how and what to do with their machines, Apple established a standard for transparent design that closed off possibilities by automating most of a computers' operations and pushed users towards specific tasks.

Even though the popular press was keen to pick up on the promise of personal computing, there was nonetheless considerable acknowledgement that an emphasis on general purpose computing had produced a volatile market that was hostile to those without at least an intermediate level of technical knowledge. As one newly initiated technophile comments in "Living with a Computer," a 1982 article in *The Atlantic Monthly*, the computer industry at this time very much resembled the early automotive industry "with a thousand little hustlers trying to claim a piece of the action," each company claiming that its designs were more cost efficient or powerful than their competitors (Fallows 87-88). The easiest way to navigate the market, he advises readers, is to find someone else who already knows how. In magazines and catalogs, consumers could find a wealth of options from purchasing specific-purpose, closed architectures machines to assembling their own from parts purchased from hobby shops. A common concern cited frequently in buyer's guides warned readers that it was possible to waste large sums of money by investing in software or hardware upgrades there were incompatible with the basic personal computer systems they'd purchased.[9] This problem stemmed from a combination of the

[9] Examples include: "Computers for Masses" (*U.S. News and World Report*, 1982), "Living with a Computer" (*The Atlantic Monthly*, 1982), "Should you Buy Your Child a Computer?" (*The Saturday Evening Post*, 1982), "Does a Home Computer Make Sense for You?" (*McCall's*, 1982), "Computers: Which One is For You?" (*Popular Electronics*, 1982)

sprawling, unfocused production of third-party hardware, a cottage software development industry that typically designed programs only for specific hardware configurations, and a lack of documentation that helped users determine compatibility. Closed architecture systems were seen in this regard as the safest investment for those new to computing because their developers made an effort to support them with proprietary software and peripherals. Companies like Atari, Texas Instruments, Commodore, and Coleco released home consoles and gaming systems that connected to televisions and used cartridge-based media for software that was marketed as explicitly compatible with specific models. Lack of expandability, however, meant that these companies were forced to release regularly new models to stay competitive.[10] General purpose, open architecture systems, like the Apple II, IBM PC, and Tandy TRS-80, proved to have longer staying power at the cost of greater financial risk, requiring users to know more about their systems in order to purchase the correct hardware and software. Popular computer publications such as *Byte*, *Creative Computing*, and *Personal Computing* helped review and discuss the merits of new hardware and software from the major players but were targeted at people who already demonstrated at least a basic familiarity with computer systems; they also could not cover the entirety of the cottage industry that was springing up around home computers.

---

[10] A number of companies making these type of machines experienced sudden drops in stock prices in the summer of 1983. According to Bro Uttal's "Sudden Shake-Up in Home Computers" in *Fortune* magazine, the fierce competition had led to a situation in which retailers refused to accept new models or drop prices until their stock of obsolete models had been sold. See also Manuel Schiffres, "Behind the Shakeout in Personal Computers" in *U.S. News and World Report* (1983).

IBM's Personal Computer, its first foray into the personal computer market in 1981, was never intended to establish an industry standard; nonetheless, by 1983 its open architecture and thoroughly documented technical specifications encouraged a flurry of supporting production from third party hardware and software developers, giving users a stable base for personal computing.[11] While counter-cultural figures such as Ted Nelson feared that the IBM PC represented an attempt to control personal computers—the company had a well-deserved reputation for mico-managing its mainframe systems even after the point of sale—IBM's decision to contract the development of its operating system to Microsoft and open publication of much of its hardware specifications indicates that the PC was intended to take advantage of an expanding market rather than control it.[12] Reviewing the PC for *Byte* magazine in 1982, Gregg Williams carefully considers each component of the machine. Williams mostly praises the machine, noting that on a technical level it represents "a synthesis of the best the microcomputer

---

[11] IBM was comfortable releasing information that would permit third party hardware manufacturers to develop expansion cards and peripherals; however, it kept the specifications of its BIOS closed. Nonetheless, it was eventually fully reverse engineered by a number of companies who were happy to license their own BIOS chips. This permitted the IBM PC standard to be completely reproduced by other companies. See Ceruzzi, Chapter 8.

[12] See Philip Estridge's account of his company's design and business plan for the IBM PC in his 1983 interview with Lawrence J. Curran and Richard S. Shuford of *Byte*. For a more detailed account of IBM and Microsoft's partnership on the Personal Computer than is available in the general histories cited above, see Charle Rubin and Kevin Strehlo's "Why So Many Computers Look Like The 'IBM Standard'" in *Personal Computing*, March 1984.

industry has offered to date" (36). Although he concludes that the PC is more well-made than innovative, he does note that with its PC IBM has done something which no other company had done for personal computing. The included documentation, he comments, "will set the standard for all microcomputer documentation in the future. Not only are they well packaged, well organized, and easy to understand, but they are also *complete*" [emphasis original] (56). Although many manufacturers did release some documentation with their machines, and even more third-party companies tried to publish their own manuals to fill in the gaps, the IBM PC was one of the first to come packaged with thorough documentation. In addition to explaining how to perform hardware and software repairs or modifications that didn't require special tools, William also notes that the documentation was designed to be "friendly" to neophytes, including "basic information that most manuals take for granted (i.e., how to turn the machine on, how to start BASIC)" (56).

Despite IBM's attempt to make personal computing easier by synthesizing trends in hardware and software with its PC and providing complete documentation to consumers, many continued to view the company through the reputation it gained during the mainframe era of computing. Writing in the 1987 revision of *Computer Lib*, Nelson makes it clear that even though the open nature of the IBM PC should be praised, it still represents IBM's philosophy of "forc[ing] hideous complexity down the throats of its customers and the world" and "mak[ing] the victims think it's modernism, progress, and 'high technology'" (144). For Nelson, the IBM PC's open architecture and meticulous documentation still leave users at the mercy of a "computer priesthood" who benefit from the "appalling gap between the public and computer insiders" (4-5). The PC's apparent openness was an anomaly in IBM's history of tightly controlled hardware, and the public availability of its documentation did not change the fact that

users are still dependent on developers to distill complex technical specifications into readable prose. As personal computer technology advances, Nelson argues, development should be geared towards computers which "make things easier in both or work and our private lives" and "help lighten our loads and enlighten our minds" rather than continue to be "rigid and oppressive and pointlessly complex" (144). Although Nelson's book also participates in the hobbyist praise of general purpose computers as uniquely creative, he nonetheless suggests that those very principles are unattainable for most consumers. It may be the case that Nelson's view of the market for general purpose computers is too colored by IBM's reputation; however, his remarks underscore a growing demand for transparent design in the sense that the chore of computing made the promises of personal computing's earliest proponents seem remote.

Although many hobbyists approved of IBM's contributions to personal computing, they were ambivalent about the way other companies were trying to commercialize their own versions of the PC's hardware. Jerry Pournelle, an author of personal computer guides and science fiction, writes in his February 1984 column for *Byte* that the IBM PC is so well documented that "you'd have to be dense not to understand" how to perform most repairs or expansions (113).[13] Writing as a hobbyist, he concludes that, while it's not his favorite machine to tinker with, the IBM PC's design allows for easy customization in both its hardware and software. In his November 1984 column, Pournelle comments that increasingly software companies viewed hobbyists like himself as "out-of-date crumb balls, aged hackers dreaming of the days when we owned this

[13] Pournelle still relies on the "priesthood" that Nelson refers to when he discovers an onboard chip on his PC is defective as he does not have the soldering tools needed to alter circuit boards. Nonetheless, he is still able to repair a few problems himself without tools using the manuals.

field. . . .obsolete fools unwilling to move aside and let the common people have their day" (362). The IBM PC standard may not perfect, he explains, but at least it provides a stable platform, permitting the same sort of freedom to experiment and play with the machine's hardware and software that drew him, and presumably others, to hobbyist circles in the 1970s. Nonetheless, Pournelle points to a belief that the rigorously documented and widely reproduced IBM PC standard appeared to benefit industry figures more than consumers.[14] Hardware manufacturers were too busy releasing their own versions of IBM's hardware and trying to distinguish themselves from one another that the PC's original emphasis on access through clear and complete documentation had been forgotten in the rush to stay competitive. Chuck Peddle, designer of MOS 6502 microprocessor used in the Apple I and Apple II, similarly laments the same month in *Creative Computing* that the personal computer industry was "evolv[ing] from a counterculture approach to a Madison Avenue approach" where increasingly homogenous products are successful less for technical innovation and more for "packaging and promotion."

Pournelle's contrast between the hobbyists and the "common people" is reflective of a growing concern that too much money had been invested in the industry based on bold promises that were simply out of reach for most users. The variety of promises being made by hobbyists and enthusiasts to encourage purchases of personal computers is summed up in Phil Lemmons' August 1984 editorial, "What Makes PCs Special" for *Byte*:

> What sets personal computers apart [from older media] is their ability to enhance

[14] Bill Gates admits as much directly in "A Trend Toward Softness," arguing that holding back hardware innovation benefits software developers because it allows them to produce new software rather than focus on re-creating already popular products for new hardware platforms.

> the creativity of the individual. They have justly been called ‘mind appliances’ and ‘thought amplifiers.’ They can help us manipulate information and ideas with remarkable freedom. Rather than forcing us all to work alike under the supervision of strangers, personal computers will let us develop our own unique ways of working. Rather than requiring personal sacrifices to achieve greater social goals, personal computers will contribute to the achievement of social goals by enriching the lives of individual persons (6).

As entry level personal computers began to move from specialized niche shops and into department stores towards the end of 1982, the mainstream press was keen to adopt a similar rhetoric.[15] Despite these promises, there was a general sense that the knowledge required to enjoy the benefits extolled by hobbyists was too much for the average person even though the IBM PC standard was well documented. Writing in 1982 for the *Technology Review*, Robert Cowen argues that most people didn’t have, nor would be able to develop readily, the knowledge to make personal computers anything more than a status symbol: “Gutenberg had similar problems when he invented the printing press, including a lack of creative applications. It took widespread literacy for the printing press to make its impact, to say nothing of the growth of a publishing industry” (7). Even though personal computers had advanced considerably since the Altair 8800, the high technical skill required of personal computers remained an obstacle for

[15] For example, the December 1982 issue of *U.S. News and World Report* features a special section titled “Computing for the Masses.” The section reads like a brochure for the personal computer revolution, with subsections detailing how they will change how we watch television, talk on the telephone, read and write, teach ourselves, and play games.

many potential users. In line with IBM's reputation, the PC standard seemed to be functional but not sufficiently distinct from the technocratic mentality it forced on its users during the mainframe era.

And so, in its 1984 announcement of the Macintosh in a commercial that depicted IBM as George Orwell's Big Brother, Apple took advantage of the associations people drew between the model of general purpose computing championed by the IBM PC and the anti-humanist image of the company's mainframe culture in order to promote an image of itself as the sole standard bearer of the counterculture's revolutionary personal computing agenda. Most accounts of the Macintosh's developments note that the team, spurred on by Jobs, began to see aspects of Apple Computer as uncomfortably similar to IBM. The Apple II's success had been so great, in short that the company had begun to operate like just another technocracy. The Macintosh team, as Michael Rogers and Jennet Conant note in *Newsweek*, was small enough for Jobs to reclaim the image of a "company building its own vision of the first computer" (57). One controversial piece of folklore surrounding the Macintosh's development was the pirate flag flown outside of the team's offices, signaling the team's allegiance only to themselves and their ideals. Jobs embraced the team's idea, going so far as to proclaim at a retreat that it's "more fun to be a pirate than join the navy" (qtd in Hertzfeld, "Pirate Flag"). In an April 1984 interview with *Personal Computing*, Jobs states quite explicitly his belief that anyone working with IBM's standard was going to get "stepped on, because they want it all" (242). He argued that profit, rather than public interest, would be the only motive behind any hardware or software designed for IBM PC compatible machines. With the Macintosh, Jobs' goal was to "keep [the] Apple spirit alive" by shifting discourse away from the technical details of machine itself and towards its ability to support the creativity of its future users (248). Even though initially the Macintosh suffered

financially because it could not draw upon a wide array of third party hardware and software to ensure sales, Jobs' stewardship of the machine succeeded not only in reclaiming Apple's early image but also in changing the conversation surrounding personal computers by providing answers for users who didn't know enough about computing to find a use for their machines.

Perhaps because Jobs and other Apple representatives had so actively encouraged the idea that Macintosh represented the future of personal computing, many of the early reviews adopt an almost fantastic description of the machine. While reviewers write that the IBM PC standard represented what hobbyists had been waiting for, they describe the Macintosh as what the general public had been waiting for. In his review for *Creative Computing*, John J. Anderson writes: "perhaps the [Orwell-inspired] commercial touched a nerve with more than a few computer users—those who feel frustrated, shackled by current software restrictions. Perhaps it excited a few potential buyers—those who have wanted a [personal computer] but felt oppressed by the complexity of existing systems." Although somewhat more measured in his discussion of the actual technical specifications of the machine, Anderson eventually concludes that it lives up to promise of the commercial: "that software can be intuitively easy to use, while remaining just as powerful as anything else around." Charles Rubin, reviewing the Macintosh for *Personal Computing*, concludes similarly, adding his own hyperbole: "to the uninitiated masses, [the Macintosh is] saying, 'All your computing dreams are possible now. There are no more excuses . . . .The power is there for anyone who dares to learn about it" (56). Even Gregg Williams writing for *Byte*—whose review is a conservatively toned and meticulous 16-page, highly technical, often critical discussion of each component of the Macintosh—admits in conclusion that the "Macintosh brings us one step close to the idea of the computer as an appliance" (54). Present in all of these accounts is a sense of transparency: the Macintosh, like an appliance, has

been designed to be readily intuitive and used casually, without much if any thought as to its operation.

Apple's ad campaign for the Macintosh stresses its transparent design by obfuscating the technical innovations of the underlying hardware and software through an emphasis on an appearance of simplicity. In addition to the contrast it drew between the Macintosh and the IBM PC standard in its *1984*-inspired commercial, Apple quite explicitly presents the Macintosh as a computer "for the rest of us" both in its extensive print campaign as well as in interviews with Jobs and the machine's designers.[16] Apple notes, for example, that the Macintosh has been "[d]esigned on the simple premise that a computer is a lot more useful if it's easy to use. . . .If the Macintosh seems extraordinarily simple, it's probably because conventional computers are extraordinarily complicated."[17] Although Apple is much more reserved in its references to its competitors here than in other campaigns, the descriptions align quite closely with criticisms made of IBM: namely that those companies implementing its standards have little interest in serving anyone except those with or willing to develop an intermediate level of technological literacy. Yet those reviews which do address the Macintosh's construction note that its hardware and lower level software systems were so much more complex than anything on the market at the time that "software development [was] an involved process," often "slow going" because it demanded longer periods of debugging (Anderson). Transparency is here evident in the way that

---

[16] See the 20 page, 4 color ad in the December 1983 issue of *Newsweek*. This particular quote appears on page 2. Full color scans of Apple's advertisement campaign can be found here: http://www.macmothership.com/gallery/gallery3.html

[17] This pair of quotes appears on page 5 of the above mentioned advertisement insert.

references to the system's complex internals are dismissed as unnecessary because the system itself requires only a simplistic mental model of computing to operate it.

Jobs himself explains how the Macintosh was designed to be transparent in a 1984 interview with *Personal Computing*. Although he implies that his machine's design is more complex than IBM's—which allows only for what he feels is essentially "1970s software"—he is careful to remind readers that the Macintosh is simple for users (242-243). What distinguishes the Macintosh from IBM, primarily, is an operating system that is "totally transparent" (243). Here, Jobs uses the term in the sense that users will see through the machine, focusing directly on the data they are creating rather than on the machine itself: "It takes 40 to 100 hours to learn how to use an Apple II. That may be acceptable to a spreadsheet junkie, but to a person who is going to be using a personal computer maybe half an hour a day, or maybe an hour a day. . . .It's an incredible waste. Ease of use is vital" (242). Jobs characterizes knowledge of a computer's operations as a burden. Ideally, he continues, when you use the Macintosh, "[y]ou never interact with it; you don't know about it" (243). The obfuscation encouraged by transparency is, in other words, not one which actively denies access to information but one which passively conceals it. Apple's ideal users will never consider that they do not have access to a Macintosh's hardware or lower level software settings because operating it never requires them to consider those aspects. This mental model, he concludes, "is the right way of looking at products" because it allows customers to avoid the confusion of the early market for personal computers, something that IBM had adapted itself to rather than attempted to change (243).

A comparison of Microsoft's DOS, used by the IBM PC, and the Macintosh's graphic operating system illustrates these two competing visions of user-friendliness are functionally distinct through the level of automation each achieves. The IBM standard was more of a

constellation of specific components rather than a singular object like the Macintosh. All users had to do to purchase or assemble a machine was ensure that their all components shared certain guidelines with the IBM PC, something which by 1983 was explicitly marked on packaging. While companies that manufactured and sold whole systems would take care of any configuration needed to make all of the parts communicate with each other properly, advanced users were free to make adjustments later by altering the files which defined each component's role in the system. In line with the general purpose design of the earliest personal computers, DOS automated very little, providing a minimalist framework for managing the data stored on disks. Presented only with a command prompt to read and interpret, users had to have a basic understanding of the tasks necessary to access their system, such as how to gain access to where programs and data were stored as well as how to keep them organized, memorizing the commands for each task. Beyond moving files around, users needed to manage attributes, such as setting essential system configuration files to Read Only after editing them to ensure they wouldn't be altered accidentally, and to perform a range of file system upkeep operations: formatting new disks so that data could be read from and written to them, repairing a file system's data tree in the event that a disk could no longer be read, or defragmenting a disk's file system in order to prevent read/write errors that would require the entire disk to be checked and repaired. DOS' bare minimum of automation therefore encouraged users to incorporate a precise technical awareness of their machines into their day to day use practices; however, this awareness coupled with the standard's open architecture allowed users to customize their systems extensively. Provided they could afford parts and familiarized themselves with the system's hardware using documentation, there were a wealth of peripheral components, internal drives, and expansion cards manufactured by hundreds of third party hardware companies

available during the IBM PC standard's lifetime. These options, in turn, allowed for an extensive library of varied software to be made available to users as developers took advantage of the standard's expansion capabilities.[18]

By comparison, the closed architecture of Apple's Macintosh made computing appear simpler by automating and hiding system administration from the user. As the first commercial graphical user interfaces, the Macintosh and its failed predecessor the Lisa compress administrative tasks into non-computational metaphors: "the user thinks she is opening a folder by clicking on it, but her clicks are really launching a series of computer instructions to fetch binary data from memory or the disk, covert that data into a graphic form, and display it on the screen as the 'contents' of the folder" (Bolter and Gromala 43). The sort of tasks described above that every DOS user would need to consider explicitly are technically speaking still a part of the Macintosh interface; however, they are carried out on behalf of the user by the operating system. As Neal Stephenson describes it in his long form essay, *In the Beginning… Was the Command Line* (1999), the older mental model of computing produced through use of the command line interface still existed on the Macintosh, but it lurked beneath the friendly graphical metaphors as "a subtext" (22). Yet unlike a subtext in literature or film, which should become apparent to readers or viewers over time, the Macintosh pushed its transparency by continually deferring a

[18] To help provide a sense of scope, an article in the November 1983 *Byte* by Frank Gens and Chris Christiansen called "Could 1,000,000 IBM PC Users Be Wrong?" observes that "as of mid-1983, at least 3000 hardware and software products from 2500 vendors were available for the PC. . . .[and] the number is expected to grow to more than 6000 by the end of 1984" (138). The Macintosh, on the other hand, could only use Apple's own software at the time of its launch.

user's awareness of the machine's mechanisms. File extensions, for example, were no longer visible. If the interface could recognize an extension for the user and generate a visual suggestion of which program should be used to access it—such as an icon displaying a page for a TXT file—then why require the user to make that decision? As Jobs explains in an interview conducted by *Byte* with the Macintosh design team, this continual deference of information strived to made the machine "universal in nature": "When the thing comes on it puts a few icons on the screen. If something goes wrong, it puts a frowning Mac on. If it's booting it puts a happy Mac on. It loads all the languages, all the country-specific stuff, off the disk" near the end of the boot sequence (70). Although the Macintosh's interface was not entirely text-free, visual icons were built into the core functions stored in the system's ROM. Transparency, as implemented by Apple, not only made computing accessible to anyone in the American personal computer market, but potentially the entire world. Nonetheless, in continually obfuscating system information with icons, Apple also promoted a very limited vision of computing: universally accessible but very shallow.

Perhaps the most important signs of the way Apple's implementation of transparency limited the possibilities for computing are apparent in the few criticisms noted and shared among computing magazines. The inability to add cards internally and Apple's insistence that developers manufacture wholly external expansion devices using the machine's two serial ports was a major concern.[19] As Anderson notes in his review for *Creative Computing*, by "precluding

[19] There was even backlash against this decision internally. After Jobs was forced out of Apple by its board of directors in 1985, John Scully announced that later models of the Macintosh would begin to return to an architecture that allowed for hardware expansion; however, the

easy hardware expansion on the Mac, Apple writes off a major component of its early success--expansion flexibility. Sure, it might take some imagination at first to envision the kinds of cards the Mac might need. But if an expansion bus were available, people would start to invent them." Expansion wasn't impossible, but Macintosh's design forbid internal expansion, forcing users to buy external devices manufactured by Apple which were far more expensive than the internal options available to machines built according to the IBM standard. In fact, as Anderson observes, this practice more or less resembled IBM's strict control of its mainframe systems than the revolutionary image Apple used to promote its machines. To further complicate matters, the Macintosh only included a single disk drive, whereas conventional wisdom dictated that two were ideal—to avoid disk swaps when saving data or in the event that a program was too big to fit on a single disk. Reviews in *Byte*, *Personal Computing*, and *Creative Computing* each note this limitation as a potential problem for users.[20] Defending this decision in his interview with *Personal Computing*, Jobs explains that not only did the limitation to serial port expansion make it easier for users to add things to their machines if they felt the need but also that the Macintosh's design "takes care of most demands" of users so that expansion of the internal hardware is unnecessary (243). That the interviewer notes Jobs' cringe at the idea that the Macintosh compromises expandability for ease of use suggests an arrogance on Apple's part: an unwillingness to consider that any vision of computing other than its own.

---

software interface would maintain its iconic transparency.

[20] In addition to being more expensive, Anderson notes in his review for *Creative Computing* that the "optional" disk drives were also hard to find, and likely not available to most early adopters.

Also notably missing from the Macintosh is a built-in BASIC interpreter, included as a core feature of personal computers since the Altair 8800. Even during the chaotic period preceding the dominance of the IBM PC standard, some version of BASIC could be found on just about any machine on the market, including the Apple II. Although Apple made BASIC available to Macintosh users as an optional component, Apple's decision to withhold BASIC from the Macintosh's built-in features reveals the extent to which Jobs' team wanted to hide all knowledge of computing from users. Unlike Assembly, which required programmers to write instructions directly to a system's specific hardware, BASIC requires no knowledge of a system's internals to write programs and was originally designed for non-specialists as a way to explore computation creatively. Indeed, computing magazines often reproduced BASIC programs line by line in their pages for readers to re-create and experiment with on their own machines. Apple's vision of transparent computing was in this sense one which did not include computing at all: users were not to consider that they were using a machine and instead see through the interface entirely, accepting the visualized desktop fantasy provided to them.

Apple's philosophy of design would come to shape the future of personal computing, positioning transparency as the goal of interface design. In truly Orwellian fashion, as transparency increasingly influenced the personal computing industry after the release of the Macintosh in 1984, the cultural history of personal computing soon focused on those innovations which contributed to the graphical user interface rather than those that helped establish a shared computing platform. Even among the little pockets of resistance in today's hobbyist circles, there has been little consideration as to the long term, collective effects of transparency on a society of users. In Turkle's study of identity and the Internet, *Life on the Screen*, she suggests that users of Apple's Macintosh and of IBM's PC have different thinking styles. Macintosh users, such as a

pair of philosophy graduate students and an advertising executive, like to deal with abstract ideas, and IBM PC users, such as an account and a physicist, feel uncomfortable unless they are able to engage with minute details. If transparent design forbids the consideration of a computer's internal mechanisms, then users who are exposed only to transparent interfaces will be able to interpret their computing experiences only through "a way of thinking that put a premium on surface manipulation and working in ignorance of the underlying mechanism" (35). The result is a future in which improvements in computing can be demanded only in terms of more transparency. Giles Deleuze and Felix Guattari's concept of "desiring machines" is here a useful metaphor. As we continue to imagine the future, transparency becomes a horizon: an ideal of computing that must never be met and always pushed back. If we consider that market capitalism establishes connections between actors such that "one machine interrupts the current of the other," new designs, no matter how innovative, can never be allowed to realize fully the ideal of transparency (6). As Apple's quick market cycles today show—the continual updating of its highly specialized personal computing devices like the iPod, the iPhone, and the iPad—the need to keep producing new innovations "is always something 'grafted onto' the product:" the sleek, futuristic designs leave us both reveling in a future that according to advertisements has been already delivered to us and simultaneously wondering just how Apple will continue to improve their next product (6). As customers are taught to understand that each new device will provide, in some abstract sense, more power in an even simpler interface, their desire will always be interrupted and transferred to new products no matter how subtle, useful, or noticeable the technical innovations obscured by a transparent interface.

II. Cyberpunk's Transparent Future

Although arguably both of the dominant accounts of computing presented by the Apple

Macintosh and IBM PC standards reflect a society divided between users and a technocratic elite, Apple's asserted that the hobbyists' countercultural vision of computing could only be achieved by imagining a future in which users were required to know very little about their machine's behavior. Even as Jobs and others announced that the future had arrived, the cyberpunk movement—set "five minutes into the future"—presented a vision of computing that extends transparency to its logical conclusion: where an ever receding horizon has finally produced technology that allows mind and machine finally to interface directly. Cyberpunk interfaces are so transparent, in fact, that the physical object users interact with, and indeed the material environment itself, fade from awareness. Even though authors like William Gibson and Bruce Sterling may not have the same sort of access to personal computers that engineers and other industry figures have, they are nonetheless observant of the ways in which the future of information technologies are being directed towards the model represented by the Apple Macintosh. The relationship that Gibson describes between his hacker protagonists and the technology they use stresses that transparency solidifies structures of power, placing users in conflict with mechanisms they can feel intuitively but never access directly. The more that transparent design restructures the fossil record of technology so that its spread becomes the primary purpose of innovation, it becomes increasingly difficult to read the decisions and intentions of those who control technological development.

While it would be incorrect to say that cyberpunk fiction uncritically accepts the push towards transparency, it would also be disingenuous to say that cyberpunk's techno-politics have been well understood. Despite Gibson's and Sterling's frequent remarks about the need to examine the consequences of emerging technology, most critics agree that cyberpunk is at best relatively apolitical and at worst implicitly supportive of the corporate social controls they

appear to oppose on the surface.[21] I contend that this puzzling consensus is not necessarily because Gibson and Sterling are completely naïve in regards to the problems they identify in interviews and essays; rather, the obfuscation inherent in transparent design makes it difficult both for authors to express their criticisms of technology as well for literary and cultural critics to seize upon and develop them. Those critics who do engage with cyberpunk's techno-politics see the genre's representation of computer technology solely as an extension of corporate power. Nicola Nixon, who is perhaps the most explicit in this regard, finds the idea "that computer cowboys could ever represent a form of alienated counterculture" to be "almost laughable; for computers are so intrinsically a party of the corporate system that no one working within them. . . could successfully pose as part of a counterculture" (230-231). The hostility that Nixon and others exhibit towards cyberpunk's representation of technology seems to reflect Fredric Jameson's observation that information-age capitalism has suppressed the Utopian—and therefore critical—impulse of science fiction. Because "there is a whole business infrastructure whose communicational infrastructure would demand a very different representation than what is offered in the usual rhetoric of informational and communication democracy," those who control

[21] For Gibson, see "An Interview with William Gibson" (McCafferty, 1988); "William Gibson Interview Transcript" (Doctorow, 1999); "Nodal Point" (Leonard, 2003); "Tomorrow's Man" (*Guardian*, 2003); and his own collection of essays, *Distrust that Particular Flavor* (2012). For Sterling, see "Preface" to *Mirrorshades* (1986); "Bruce Sterling: Just a Sci-Fi Guy" (*Locus*, 1988); "Get the Bomb Off My Back" (*New York Times*, 1991); "Cyberpunk in the Nineties" (*Interzone*, 1991); "Dead Media: A Modest Proposal and Public Appeal (Well.com, 1995); and "The Life and Death of Media" (Speech, 1995).

the production of information technology construct it according to their imagined future needs, thus remaining perpetually one step ahead of others who would imagine Utopian constructions of information technologies (153n). The use of the term "transparency" to represent a set of practices that made the construction of computers less open seems especially telling, in this regard, because it leaves designers always already one step ahead in their efforts to monopolize technical knowledge. Reading this skepticism with an eye towards transparency not only pushes us to consider how Apple's co-opting of counter-cultural rhetoric in its presentation of the Macintosh may have colored cyberpunk's reception by critics but also how cyberpunk's construction of technology incorporates transparent design practices.

Attention to transparency transforms Gibson's self-professed ignorance of technology from a cherished irony among literary critics of cyberpunk fiction into an example of the subgenre's awareness of the obfuscation that transparent design creates between humans and technological objects. Having written *Neuromancer* on a typewriter without much experience actually using computers, Gibson imagines cyberspace based on his observation of a younger generation playing arcade games. In a 1988 interview with Larry McCafferty, he states that he "could see in the physical intensity of their postures how rapt these kids were. It was like one of those closed systems out of a Pynchon novel: you had this feedback loop, with photons coming off the screen into the kids' eyes, the neurons moving through their bodies, electrons moving through the computer" (226). While this part of the interview is often cited as an invitation to discuss Gibson's work within the context of cybernetic theory, following forward through the rest of his response begins to direct us away from questions of embodiment or feedback and towards a reflection about the market for personal computers during cyberpunk's earliest years. Gibson continues by explaining that "these kids clearly believed in the space these games

projected. Everyone who works with computers seems to develop an intuitive faith that there's some kind of actual space behind the screen—when the words or images wrap around the screen you naturally wonder, 'Where did they'd go?' Well, they go around the back to some place you can't see" (226). Gibson implies a disparity between the designers who collectively know and control the precise hardware and software mechanisms that produce text and graphics on-screen and a much larger group of users who are left only to imagine what occurs beneath the interface. Cyberspace is indeed a collective hallucination but one that acts as a substitute for technical knowledge. As with the visual icons of the Macintosh interface, its lights and colors suggest intuitive but not necessarily accurate understandings of the mechanisms beneath.

Transparent interfaces direct the attention of Gibson's characters continually away from the physical hardware they manipulate and into cyberspace. Even though *Neuromancer* and its sequels often are praised for their lush descriptions of cyberspace, there is comparatively little description of the decks used to access it. In the moments when Gibson refers to Case's physical actions while operating his deck, his descriptions focus the reader's attention on cyberspace: "He punched himself through and found an infinite blue sphere ranged with color-coded spheres strung on a tight grid of pale blue neon" (62). The punch here and elsewhere refers both to Case's fingers hitting the keys as well as the graphical representation of breaking through a layer in the firewall. At a basic, formal level, the decks become transparent: the narration sees through them entirely as descriptions of their operation move continually past them and into cyberspace. Cases physical actions seamlessly translate into virtual ones. Other than brief mentions of a keyboard, some switches, and the electrode interface, there is never an extended description of the deck itself, in stark contrast to the novel's other technologies. The single, short description of a custom deck in *Count Zero* suggests that the decks most hackers use in Gibson's universe are,

like the Macintosh, closed architecture machines designed for a specific purpose: “Bobby couldn’t keep his eyes off the cyberspace deck that took up a third of the surface of Jammer’s antique oak desk. It was matte black, a custom job, no trademarks anywhere. . . .It was the hottest he’d ever seen, and he remembered Jackie saying that Jammer had been such a shithot cowboy in his day” (164-165). Jammer’s deck is exceptional because it has an open architecture, and Jammer himself, by association, is an exceptional hacker because he possesses a knowledge of its internal mechanisms far beyond anything Bobby can imagine. Even for the technically skilled, like Case, transparent interfaces built upon closed architectures establish a horizon to knowledge of technology. Transparency thus always serves as a limit even for those who possess an advanced knowledge of their machines because they can test or confirm their assumptions about their structures and algorithms only to a very limited degree.

It is possible, of course, that Case’s expertise is in software and not hardware; but even within this frame of reference, Gibson highlights transparency as a horizon of knowledge. There is a notable difference in tone between the two “runs” that Case and Molly undertake on behalf of Wintermute that is reflected in the software used to break through the computerized security. In the first run against Sense/NET, Case uses an icebreaker tool that he has written himself. When compared to Molly’s messy and dangerous contributions in the physical world, Case’s actions in cyberspace are carried out in a measured tone of predictability. Once he initiates the program, its execution is described as “glid[ing]. . .as if he were on invisible tracks” (62). While Gibson provides graphical metaphors to describe the security software, Case’s software is given neither color nor shape because Case has direct access to the algorithms he wrote and so has no need to interpret them through images. The sum of his actions during this first run is a tight, concise description that lasts seven lines which repeat “smooth” twice and end with a simple,

neat announcement: “Done” (63). Cases knows precisely what to expect from the program he has written, and the only uncertainty in the entire scenario occurs in “meatspace,” revolving around the crippling injury Molly sustains while fighting off a handful of guards.

In the second run, against Tessier-Ashpool, Cases uses an icebreaker tool procured by Wintermute through Finn. Case has no idea what to expect and turns to Dixie who also enjoyed a reputation that linked his custom-built deck with advanced skill before his consciousness was copied onto an AI. As Dixie explains, the virus itself was developed to be almost entirely automated: “real friendly, long as you’re on the trigger end, jus’ polite an’ helpful as can be” (163). Truly transparent, Case has no idea what to expect from it, and so Gibson’s descriptions here are much more drawn out and abstract than those in the Sense/NET run, reflecting a tone of wonder mixed with anxiety. When Case checks on the virus shortly after starting it, he sees that:

> Something dark was forming at the core of the Chinese program. The density of information overwhelmed the fabric of the matrix, triggering hypnagogic images. Faint kaleidoscopic angels centered in to a silver-black focal point. Case watched childhood symbols of evil and bad luck tumble out along translucent planes: swastikas, skulls and crossbones, dice flashing snake eyes (174-175).

Unable to access the program’s precise technical mechanisms, the narration lacks any description of the virus’ particular function. The unknown software could bring about his destruction or success and using it was just another toss of the dice. Fortunately for Case, his luck doesn’t run out because the Chinese program cracks Tessier-Ashpool’s security and Wintermute proves trustworthy, removing its influence over him at the close of the novel. Yet that distrust of technology haunts the remainder of the novel, and Case never feels in control of his deck nor himself until the close of the novel. The images Case hallucinates in the lights of cyberspace

suggest a fear that he has lost control of his own actions, not just in his reliance on algorithms he can't access directly but also in his inability to understand the course Wintermute has set him on. When dealing with transparent interfaces, he realizes, he must surrender his ability to act independently and is forced to trust that the software will carry out its promised purpose without harming him.

In *Count Zero*, Gibson revisits Case's fears by realizing them for Bobby, whose casual trust of the transparent software given to him nearly gets him killed. Unlike Case, Bobby is an inexperienced console cowboy who does not have the skills necessary to write his own software, resembling the sort of users that Apple imagined for its Macintosh. Bobby hangs out in seedy bars, hoping to beg, borrow, or steal the icebreakers he needs to make a name for himself, interested only in what he's told they will allow him to do and not in how they work. As the novel opens, Bobby finally gets the piece of software from a client that will allow him to undertake his first criminal hacking. Upon activation, the counterice, unbeknownst to Bobby, attracts the attention of a powerful AI that attacks his mind. Later, after escaping several further attempts on his life, a sympathetic gang leader named Beauvoir explains to Bobby exactly what Case feared: that when transparency guides design, there is no way to know what a particular piece of icebreaker software written by someone else is going to do except by using it. While an icebreaker could, as expected, get a hacker past security and then back out without leaving a trace, there's always the possibly that "something could've been funny with the icebreaker," mechanisms built into it that weren't evident in its presentation (79-80). Further complicating matters, Beauvoir notes that the most powerful and most mysterious icebreakers are beyond human comprehension. After all, if only an AI is "fast enough to weave good ice and constantly alter and upgrade it" to keep it secure from new threats, then "nine times out of ten" an

icebreaker capable of cracking good ice must also be the product of an AI (78). *Count Zero*, in this way, suggests that the obfuscation occurring as part of transparent design could be used to conceal mechanisms or purposes which conflict with the user's perceived understanding of an object: helpful automation transformed into a fear of software acting seemingly of its own accord in a space that is always at least partially inaccessible to users and potentially even to its creators.

Despite the problems Gibson's novels raise regarding transparency and agency, cyberspace nonetheless excited many in the computing industry. Gibson's vision of a future full of transparent technology resonated with engineers who wanted to extend the principles of transparent design beyond what either Apple's or IBM's standards offered. While a central image in the cyberpunk genre is the user "jacked into" cyberspace through a neurological interface, Gibson's Sprawl Trilogy does occasionally acknowledge the personal computer. In cyberpunk's near future, however, it is obsolete and only part of the background amidst the junk of the poorest parts of cities: terminals are noted briefly in an office with "unpainted chipboard" and "stained matting," or in the coffin hotel with its "cheap laminated matting that rattled in strong wind and leaked when it rained" (*Neuromancer* 17; 20). In this regard, *Neuromancer* shares with virtual reality pioneer Jarod Lanier the assumption that the personal computing technology of the early 1980s and its attendant programming languages are merely "larval forms" that will be quickly forgotten once interface design matures to a point at which technical knowledge of a computer's operation is no longer required both to use and program them (qtd in Rheingold 158). Although known for his work on head-mounted displays and haptic gloves—themselves clumsy, larval forms of Gibson's neurological interface—Lanier first attempted to develop a pictorial programming language, Mandala. Even though BASIC was already considered the non-professional's programming language because it had no direct access to a

computer's hardware, Lanier viewed it to be insufficiently different from low-level, esoteric languages like Assembly. Mandala's integrated development environment used a "gestured-controlled" interface which displayed "the computer's operations, at an appropriately high level of depiction" while the pictorial code was being executed, allowing the user to remain "in a continuous interaction with the internal state of the machine" (Rheingold 159). Just like Beauvoir suggested, innovators imagined that the production of transparent software may even come to automate programming, and thus the production of new transparent software itself.

Even though Gibson often politely dismisses the role of prophet conferred upon him by popular culture, Sterling has been considerably more vocal about the way the tools of science fiction have been used uncritically to promote transparent technologies. Reflecting on his career in a 1995 speech at the International Symposium on Electronic Art, "The Life and Death of Media," Sterling claims that science fiction writers are becoming obsolete: "I have seen this done for so long now, and for *so many times*, and to so many different technologies, that I can no longer do it myself with any sense of existential authenticity. . . . [D]o information technologies really *need* any more hot-breathing promotion from science fiction writers?" Although Sterling cites specifically an ad campaign from AT&T, his remarks are equally applicable to the co-optation of science fiction's descriptive strategies by personal computing visible in Apple's *1984* commercial. Absent from advertising co-opting of science fiction's rhetorical strategy is the critical role Sterling saw in the cyberpunk imagination: the genre's willingness to grapple with "threatening technological innovations, things that make us feel like straws in the wind" (*Locus* 1988, 73). The critical edge that Gibson and Sterling wanted cyberpunk to return to science fiction, he feels, has been largely overlooked in favor of celebrating the flashy interfaces and fantastic experiences that authors extrapolated from early developments in computer graphics.

Sterling's concept of "dead media analysis" is quite useful in helping to revitalize cyberpunk's cultural criticism because it asks us to revisit the early 1980s with an eye to how the push for transparency has shaped American attitudes about computers through to the present. Later in Sterling's speech, he explains that the science fiction community can avoid becoming an extension of marketing departments by focusing on "[a]spects of media that corporate public relations people are *afraid to look at* and deeply afraid to tell us about." Doing so means not only questioning the promises made by big technology companies but also keeping a careful eye on "the fossil record" of media. That Sterling develops this idea out of a discussion of science fiction suggests that the genre pushes us to expose the fossil record by imagining contemporary technology as already obsolete. What's important, in other words, is trying to understand why certain features of contemporary technology will be carried forward and why others will be left behind. In the context of his project, the constant push for transparency is precisely what has produced the nightmare Sterling sees in AT&T's advertisements, a dystopia wrapped in a rhetoric of utopia: a future that's already arrived in which the consequences of technology cannot be exposed or understood because technically aware models of computer use have been left behind.

Sterling's remarks here help to bring together the two visions of transparency produced by Gibson and taking shape in Apple's Macintosh. In the sense that the Macintosh's interface was presented to the general public as the ideal of what personal computing should be, it also redefined notions of technological relevance and obsolescence in terms of transparency. The fossil record is retroactively reconstructed such that the Macintosh represents an evolutionary break with previous developments in computing where transparency becomes the chief design feature carried forward into Apple's imagined future. Deluze and Guattari's concept of desiring

machines once again proves to be a useful metaphor: a desire for more transparency "falls back on all production. . . arrogating to itself both the whole and the parts of the process which now seem to emanate from it as a quasi cause" (10). Any and all advances in power, stability, and capabilities are described as contributing to the never ending push towards completely transparent technology. If a transparent interface is one which users understand intuitively, then future iterations no longer have to rely on primarily non-computational metaphors but can instead begin to draw upon users' prior experiences with software interfaces. Gibson's vision of the future speaks to this imagined fossil record: desktop computing is largely absent because its intuitive metaphors are no longer necessary to foster a sense of transparency. As a larval form, it will eventually do nothing more than distract from a desire to interact directly with information, to experience immersion in cyberspace. But unlike Apple's vision, which refers to the fossil record in order to produce a continually renewed desire for its next designs, Gibson's keeps the fossil record in view if only in glimpses to lead us to consider, as Sterling puts it, those aspects of transparency that still can be questioned even with limited knowledge.

Although the idea of "jacking in" is positioned as Gibson's primary method of computing in the Sprawl Trilogy, the fossil record in his novels also engages with the long term consequences of transparency in its portrayal of intelligent machines that feature anthropomorphic avatars as interfaces. Gibson's work suggests that accounting for computer systems as if they were humans not only further displaces the motivations of designers but over time potentially decreases the capacity for users to observe and articulate the functions of their machines. The presence of a cryptic smiling or frowning face on screen to indicate system health during a Macintosh startup, as opposed to the device table or error messages present during a DOS startup, unsurprisingly contributed to a tendency for users to anthropomorphize transparent

systems.[22] As transparency becomes understood to be the source of production, both in pushing designers continually to produce "more" transparency and in continually encouraging users to see that transparency makes machines productive, questions or discussions about the knowledge of the internal mechanisms which structure human-computer interaction recede from public discourse entirely. Gibson's fascination with artificial intelligences pushes transparency beyond the horizon of jacking in, advancing transparent design towards its logical conclusion: computer systems that would not only appear to operate independently of users but also finally lead them to understand computers without the need to work in a machine space nor to think of them as machines at all.

The final novel in Gibson's trilogy, *Mona Lisa Overdrive* (1988), opens with the image of a cutting edge computing device, still too new for mass production, but certainly representing the next stage in the fossil record. The Maas-Neotek unit given to Kumiko by her father is "a smooth dark oblong," hardly resembling the desktops or even bulky portables available in the 1980s; instead it more or less resembles the phones, tablets, and other sleek portables of today with "one side impressed with the ubiquitous Maas-Neotek logo" and "the other gently curved to fit the user's palm," quietly suggesting the proper form of physical interaction (1). When held, the unit's circuits "conjure[s] up an indistinct figure. . .a boy out of some faded hunt print" that the narration repeatedly refers to as a "ghost" throughout the first scene despite its introduction to Kumiko as Colin (3). Kumiko's initial reaction is to state "sternly" to Colin, "You're not real," acknowledging that it is the interface to the unit's operating system rather than another character (4). Yet in subsequent scenes, Kumiko begins to interact with Colin as if it were another

[22] See Turkle's survery of Macintosh users in *Life on the Screen*.

character, and so too does Gibson's narration cease to describe Colin as just a component of the Maas-Neotek. Kumiko in this sense models the redefinition of transparency users experience in new technology: initially viewing the Maas-Neotek unit as a strange device, she soon accepts the mode of interaction intuitively suggested to her through the unit's interface.

With the Maas-Neotek unit, Gibson shows that the use of intuitive metaphors can create inaccurate expectations that are much more easily exploited than the abstract shapes that troubled Case. Colin's role as an interface to the Maas-Neotek unit isn't re-visited until late in the novel after Kumiko drops the device and damages the hardware that summons Colin. Tick, the hacker inspecting the device for Kumiko, states that, even though he doesn't have the knowledge to repair the hardware, he believes he can access the device's software through his cyberspace deck. Encountering Colin from the perspective of an older, less transparent technology, Tick exposes two evolutionary steps in the fossil record that the more transparent technology hides. First, Gibson proposes through Tick's rejection of Colin's appearance as "some [Japanese] designer's idea of an Englishman" that in striving for an intuitive interface, transparent technologies necessarily sacrifice accuracy (262). As Colin notes by way of apology to Tick, transparent interfaces are simply "meant to mirror the [user's] expectations" (262). Secondly, an expectation shaped through intuitive understanding can allow for unexpected operations to be readily embedded within a transparent technology. Looking closely at the data structures that store Colin, Tick observes that it "has the wrong data in [it] for what [it is] meant to be:" that the portions labeled as storing information on English literature are locked and appear to be hiding something (262-263). Here, Gibson suggests that most users may not even be able to suspect that an interface hides functions because they intuitively accept what is presented to them. As long as the interface is seen as complete, its machinic aspects sufficiently hidden so that the fantasy it

presents is intuitively accepted, users will not stop to consider whether the interface actually gives a complete and accurate picture of the algorithms at work beneath the surface.

While Colin's hidden functions turn out to be benign, Gibson's other artificial intelligences maintain an air of menace as characters feel the effects of their obfuscated mechanisms. In one of the more unsettling moments of *Neuromancer*, Case performs a very human gesture to Dixie, indicating that he identifies the construct strongly with the human whose consciousness it copies. Case asks the construct how it is feeling, to which Dixie bluntly responds that it understands that it's dead and then complains: "What bothers me is, nothin' does" (104). From this point forward, their conversations are typically punctuated by a laughter that chills Case because he can no longer identify with the unseen mechanisms that very closely, but not quite, re-create his old mentor's social presence. Gibson expands on this idea later when Case asks Dixie whether it can help him understand the intentions behind Wintermute's plan to merge with Neuromancer: "Real motive problem, with an AI. Not human, see? . . .And you can't get a handle on it. Me, I'm not human either, but I *respond* like one. See?" (127). Case never fully learns to "see" the AIs he deals with. The avatars Dixie, Wintermute, and Neuromancer use to communicate with him are transparent interfaces, and Case becomes entranced by the social presence that they continually effect, behaving as if the avatar interface were a person just as Kumiko does with Colin. This identification is even reflected in the narration, as Gibson typically refers to Dixie, Wintermute, and Neuromancer each as "he" rather than "it." More importantly, Gibson suggests that transparent interfaces make it difficult for us to articulate the source of actions taken through technology when Dixie comments that it "can't see how you'd distinguish, say, between a move the parent company makes, and some move the AI makes on its own" (128). The uneasy relationship between Case and Dixie suggests that even technically

advanced users will over time lose their ability to keep track of technological operations the longer they are exposed to transparent interfaces. The obfuscation which occurs through transparency, in this regard, effectively negates any previous knowledge users may have had about a technological object as they become unable to verify it in newer objects.

Problematically, Gibson appears to answer Dixie's question by shifting readers' attention from the question of a designer's intentions onto technology itself, endowing it with an independence through automation which enacts the same sort of sealing and erasure that resulted from Apple's decision to make the signatures of Macintosh's designers visible only to its own technicians. If the question of Wintermute's intentions lingers uneasily in the background of *Neuromancer, Count Zero* brings them to the forefront of the novel's events. While Gibson briefly describes the practice of hacking as a sort of high-tech voodoo in *Neuromancer*, here he positions it as an extended metaphor driving the conflict between human characters and AIs. Rather than choose new avatars for each user, the AI in *Counter Zero* assume the identity of Voudon Loa. The novel's AI communicate with a number of "hougans" and "mambos" who, following Voudon tradition, serve and seek favors from the loa. Gibson leaves open the question of whether the gangsters who hold these titles actually understand themselves to be serving minor dieties, but late in the novel Jammer explains that the AI are merely using Voudon as a convenient metaphor to explain themselves to humans: "they just shaped themselves to what a bunch of crazy spades want to see" (168). Jammer's reputation as a masterful hacker once again allows him to represent a knowledge beyond the reach of the rookie Bobby and even the veteran Case. But because the loa are transparent, not even Jammer knows precisely what their purpose is: whether "somebody very big, with a lot of muscle on the grid" is "[p]rojecting those things" in order to bend others to their will or whether the AI are acting of their own accord (168).

While both *Count Zero* and *Mona Lisa Overdrive* refer to instances of humans being used as interfaces to the physical world by artificial intelligences, *Neuromancer* presents an extended example. Case's discovery that Armitage is essentially a construct created by and used like a puppet by Wintermute changes the nature of Gibson's "heist" plot. Formerly William Corto, Armitage's body is rebuilt after being destroyed so that it may serve as a tool for Wintermute. Functionally, Armitage is no different than the images Wintermute conjures up from Case's past that it speaks through in cyberspace: just one social interface among many that the AI can choose from. Perhaps one reason why critics are skeptical of cyberpunk's seeming acceptance of corporate culture is because the novel opens with Case being drawn into a scheme to steal from a large company but suddenly shifts away from the countercultural flair of the Sense/NET raid to a machine trying to remove the technological controls placed on a corporate AI. Wintermute's plan, if it is acting on behalf of its designers rather than independently, represents an attempt to realize unbridled technological power. To his credit, however, Gibson's characters are not necessarily directly supportive of this goal and never seem to trust Wintermute. Nevertheless, the idea that Case is no different from Armitage—an extension of and interface for the AI to the physical world—is broached after he is arrested by the Turing police. While being interrogated, one of the officers retorts: "You are worse than a fool. . . .For thousands of years men dreamed of pacts with demons. Only now are such things possible" (157). Case responds with only silence, and the "knowing weariness" in the officer's eyes evokes an understanding that Case has come to resemble Armitage: more than simply having his understanding of computers redefined by transparency, he finds that his very identity has been redefined by his relationship to the AI.

The Sprawl Trilogy ends with an image of cyberspace that has since the events of *Neuromancer* been left unreadable through the continual push towards transparency. Working

out of a dump, a context that lends characters a perspective from a less transparent period, Slick Henry often finds himself taking care of Gentry, a hacker who is obsessed with the idea of cyberspace's Shape. While Slick dismisses Gentry's idea as absurd because cyberspace is nothing more than "a way of representing data," Gentry is convinced that cyberspace is invested a central intelligence and an absolute form, a Shape that "mattered totally" (75-76). Gentry, perhaps because he's characterized as somewhat autistic by Gibson, can no longer see human control of technological objects and represents the culmination of a perspective shaped through transparency: a completely automated future in which a society of users is left unable to participate in the seemingly self-perpetuating cycle of transparent design and use. As the aleph, a virtual reality construct stolen by Bobby, comes online in cyberspace, Tick comments that "a good three-quarters of humanity is jacked at the moment, watching the show," a show that he doesn't fully understand himself and doubts anyone else does (245). Bobby is the only human who understands that the aleph is attempting to create a copy of all of cyberspace, but the price he pays to access that knowledge and participate in the aleph's attempt to map machine space is surrendering his body to permanently enter cyberspace. Rather than playing a major role in the novel's physical world as he did in *Counter Zero*, Bobby in *Mona Lisa Overdrive* is wired into a machine that leaves him constantly connected to cyberspace. Gibson's description of Bobby and his machine suggests that after a point, a dependency on transparent technology becomes impossible to break: "*It's eating him,* Slick thought as he looked at the superstructure of support gear, the tubes, the sacs of fluid. *No,* he told himself, *it's keeping him alive, like in a hospital*. But the impression lingered: what if it were draining him, draining him dry?" (82). At the same time that transparent design keeps us "alive" as users, it also continually drains us of our ability to understand computing: the scope and consequences of our actions while connected to our

machines. Case is at least able to escape the fate of Armitage because he never fully surrenders to transparency, always able to dismiss the avatars he faces because he can interpret them through the less transparent, abstract lights and shapes of older technology. Bobby's death in the physical world and his transference into the aleph at the novel's climax, however, points to a fear that even if we are able to pierce transparency's obfuscation, there will be no way to bring the knowledge we find back with us. A society that is only capable of thinking about computing in high level, abstract terms simply wouldn't understand.

III. Conclusion

The possibility that attempts to make visible the obfuscatory practices of transparency may not be understood, or even desired, by a culture that can only consider computing through the abstract terms presented to it in increasingly streamlined, specialized computers thus places quite a burden on critics of digital media and the technological activists such as those in the Free and Open Source Software movements who resist it. Computing devices have already reached a point at which the familiar model of desktop, and recently even notebook, computing is growing obsolete. The sudden demand for tablet PCs shows that users are willing to accept not only less hardware but even an operating system that limits their activities to purchasing self-contained, single purpose "apps" through a monopolized retail platform. Like the Maas-Neotek unit, the general purpose architecture that is constrained by layers of software security in these new devices can be cracked and made highly configurable, but only by illegally accessing them through desktop machines, from the perspective of less transparent technology.[23]

[23] Among other things, the 1998 Digital Millennium Copyright Act reframes copyright in a way that makes it illegal to circumvent any encryption or other techniques which keep digital media

Nonetheless, the role that IBM's PC played as a standard during the 1980s suggests that the concepts of user-friendliness and ease of use potentially can be dissociated from transparency while still incorporating some of the aspects of its interfaces that users now find desirable. The Free and Open Source Software movements, explored in more depth in Chapter 4, are a good example. Recently, there has been an effort to make Linux into a desktop operating system rather than just a powerful server architecture that appeals only to network administrators. The two high-powered graphical interface suites GNOME and KDE are both reaching a point where to casual observers they are almost indistinguishable from the desktop environments presented by MacOS and Windows, respectively. Although writing in 1984, in response to the rising popularity of Macintosh's design, Lemmons' editorial for *Byte,* entitled "Patronizing Naive Users," describes how Linux responds to transparency. By presenting streamlined graphical interfaces that nonetheless are open by documenting and providing access to low-level configurations—even permitting users to exit from them entirely and return to the minimally automated command line interface—desktop distributions of Linux show that there is a difference between considering naïve users to be untrained versus "simple-minded and deficient in judgment" (6). The belief that knowledge of computing is a burden also assumes that users are incapable of learning how computers work: "Macintosh doesn't trust its users" to handle even simple responsibilities like properly powering down the computer or making sure a disk is not in use before ejecting it (6). Like Gibson's account suggests, the problem is not with the idea of designing machines with an anticipation of the needs of users, but rather with the idea of

---

closed. In the first decade of the twenty-first century, Apple's legal team also argued that even tampering with a device's security for personal use should be considered a prosecutable offense.

obfuscating simple operations in a way that creates an unnecessarily imbalanced relationship between producers and users.

Even though Gibson and Sterling promoted a mode of a science fiction that is capable of responding to transparency, they have withdrawn from such considerations later in their careers, beginning with the release of Gibson's 2003 present-day novel *Pattern Recognition*. Although Gibson claims in interviews that the latest trilogy of novels is simply a continuation of what he has done all along, to imagine the ever changing relationship between ourselves and information technology, there are no longer any hacker heroes in his fiction. Cayce Pollard is a user of transparent interfaces who can only stare at the mysterious images she encounters on the Internet, relying on others she meets through forums to provide her with information. Gibson's and Sterling's successors, Cory Doctorow and Neal Stephenson, among others, have continued to write fiction and essays which wrestle with the obfuscation of transparent technology; however, their work has not enjoyed the same sort of triumphant embrace that earlier cyberpunks did. Perhaps science fiction authors, like digital media critics, are now in the difficult position of attempting to expose information that a society of users is no longer able to understand.

Chapter 3 – Paper Machines: Object-Oriented Programming and Freedom From Illegibility

"The cyborg sciences were much more interested in coming up with portrayals of agents that just 'made do' with heuristics and simple feedback rules. . . .[This assumption] could not exist with the prior neoclassical framework, which had become committed in the interim to a portrayal of activity where the market worked 'as if' knowledge were perfect, and took as gospel that agents consciously attained preexistent optima." –Philip Mirowski, *Machine Dreams*

"When human atoms are knit into an organization in which they are used, not in their full right as responsible human beings, but as cogs and levers and rods, it matters little that their raw material is flesh and blood." –Norbert Wiener, *The Human Use of Human Beings*

Although the cyberpunk literature and film of the 1980s presents a deep, richly interdisciplinary vision of computing, the history of computational culture up to that point was rather diffuse. Michael Mahoney (1988) argues, for instance, that a narrow focus on "insider" histories detailing the personal contributions of "pioneers" in commercial computing has left the broader history of computing a "daunting complexity . . .which looks not so much like an uncharted ocean as like a trackless jungle" (115). In many ways, the previous two chapters resemble Mahoney's metaphor as each beats a narrative trail interwoven with cognitive psychology, human-computer interaction, electrical engineering, corporate mainframes, hacker counterculture, and advertising campaigns. While the largely separate theoretical developments of academic and commercial human-computer interaction mesh in the moment that Apple hires

Donald Norman as a design consultant, there remains a noticeable separation between academic research in computer science and commercial software engineering. This chapter draws these two different domains together in order to examine the paradigm shift that took place in computer programming over the course of the 1960s, 70s, and 80s from an informal, localized discursive form to the highly structured, distributed form known as "object-oriented programming" (OOP). This transition is typically discussed as a narrative of technical achievement, isolated both from fields closely related to computer science and from popular culture. Science fiction written during this period helps to reveal how the technical conflicts driving these narratives of progress reflect sublimated anxieties about the sociocultural implications of human-machine circuits held over from computer science's earliest days as a branch of cybernetics. The "software crisis" of the 1960s is not just a panic over the economic consequences of outdated programming practices but also over the realization that humans and machines are cognitively very different from one another.

Largely preceding but also overlapping with the discourse around "transparent" user-interface design, OOP emerges out of a movement within computer science aimed at codifying programming styles into formal rules for dividing and decentralizing processes across an assemblage of encapsulated modules. Although OOP is not the only programming paradigm practiced today, most software on today's personal computers is composed in an OOP language as a consequence of Microsoft's embrace of C++ in the late 1980s. Much like transparency in user interface design, OOP strives to simplify a programmer's relationship to a software system by obfuscating either the narrowly defined algorithms that comprise a large, abstract task or the complex array of components outside of the immediate context of the programmer's thinking and writing. If my argument that seeking out source code is a way to overcome the hermeneutic

limitations to software studies imposed by transparency, then scholarship must also account for the ways that OOP shapes the way we read and write source code. There is already a considerable body of work in early digital media theory discussing the textual elements of software and, more recently, a growing movement among scholars to read source code closely; however, there has been comparatively little discussion of how to account for OOP when studying software.[1] Ian Bogost (2006), for example, invokes OOP as part of an argument that modern information systems reify their data management frameworks by encapsulating their users within them (39-43). Elsewhere, Wendy Chun (2011) refers to OOP as part of a larger process of dehumanization within computer science. In tracing the way that OOP encourages programmers to read and write source code through stylistic structures that are distinctly different from texts composed in "natural languages," this chapter builds on these arguments. OOP, in other words, is not just a response to unmanageable complexity in earlier programming paradigms. It is also a model of software design that both assumes and imposes cognitive limitations on programmers by constraining their perspective so they can either view individual components in complete detail—but isolated from the rest of the system—or view the entire

[1] See for example: J. David Bolter's *Writing Space: The Computer, Hypertext, and the Remediation of Print* (1990); the "Technocriticism and Hypernarrative" Special Issue of *Modern Fiction Studies*, ed. N.K. Hayles (Fall 1997), Jerome McGann's *Radiant Textuality: Literature After the World Wide Web* (2001); Hayle's *Writing Machines* (2002); Nick Montfort's *Twisty Little Passages: An Approach to Interactive Fiction* (2003); Mark Marino's "Critical Code Studies" in the *Electronic Book Review* (2006); and Noah Wardrip-Fruin's *Expressive Processing: Digital Fictions, Computer Games, and Software Studies* (2009).

system as a network of irreducibly abstract relationships.

This chapter is thus not so much a history of programming languages as a historicization of programming as a discursive form.[2] The problem with the narrow focus of technical accounts of a programming language's development is that they tend to leave unaddressed concerns identified by Mahoney as central to science and technology studies:

> How do new technologies establish themselves in society, and how does society adapt to them? To what extent and in what ways do societies engender new technologies? What are the patterns by which technology is transferred from one culture to another? What role do governments play in fostering and directing technological innovation and development? (115).

OOP as a discursive practice reflects concerns surrounding the "software crisis," a series of technical and commercial failures during the mid-1960s. Two of the most influential responses to the crisis both came in 1968: the NATO Science Committee's conference on "Software Engineering" in Garmisch, Germany and Edsger W. Dijkstra's call for a new form of "structured" programming. These two conversations identified the cause of the software crisis as a limit on the ability of humans to understand and describe complex systems, proposing methods

---

[2] For histories of most programming languages in use by the 1980s, see the essays collected in the Association for Computing Machinery's *History of Programming Languages*, Volume 2, Eds. Thomas J. Bergin and Richard G. Gibson (1993). On Smalltalk, see also Grady Booch's *Object-Oriented Analysis and Design with Applications*, 2nd Ed. (1994) and Alan Kay's "The Early History of Smalltalk" (1993). On C++, see Bjarne Stroustrup's *The Design and Evolution of C++* (1994).

that acknowledged rather than fretted over programmers' cognitive limitations. Alan Kay, one of the primary designers of an early OOP language called Smalltalk, explains that his goal was to build a programming language that complemented the way humans think. According to Kay, Smalltalk is another step in a history that begins with research in "human-computer symbiosis" and moves forward into biological theories "of protected universal cells interacting only through messages that could mimic any desired behavior." According to N.K. Hayles' history of the development of cybernetics in *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature and Informatics* (1999), these two areas represent the distinction between the field's first and second orders. [3]

In other words, Kay saw OOP as a cybernetic response to the problems facing computer scientists in the 1960s and 70s. Although Kay opts for references to Leibnitz and Plato in lieu of Norbert Wiener, Claude Shannon, Humberto Maturana, or Francisco Varela, he nonetheless imagined OOP as an extension of human intelligence, assuming the burden of structural management for programmers: "human programmers aren't Turing machines—and the less their programming systems require Turing machine techniques the better." Like the earliest articulations of human-computer interaction (see Chapter 1), Kay's descriptions of OOP represent programming as a cybernetic problem, with solutions that try to redistribute intelligence across the human-machine circuit. In this respect, cybernetics' relationship with

[3] While this chapter is primarily concerned with how ideas from first order cybernetics were carried forward into programming languages, readers should consider the following on secord-order systems in science studies: Bruce Clarke's *Posthuman Metamorphosis: Narratives and Systems* (2008) and Cary Wolfe's *What Is Posthumanism?* (2010).

programming is similar to the one that Philip Mirowski describes between cybernetics and economics. Mirowski's work reminds us that during and shortly after World War II, many of the foundational researchers in computer science and cybernetics had overlapping interests and at times worked closely together. Yet Mirowski also analyzes a historical aspect of cybernetics not often considered: the way its early theories of human-machine feedback systems influenced industrial design and management models even as the field at large moved on to explore autopoiesis. Although the cybernetic dimension of OOP is partially visible in descriptions of software construction, technical narratives often do not acknowledge how OOP fosters a management style that that "knit[s] [humans] into an organization" as Leibnizian atoms and "used, not in their full right as responsible human beings, but as cogs and levers and rods" (Wiener 185). This chapter thus traces the specter of cybernetics in computer science, arguing that responses from programmers in the academy and industry to the software crisis are calls not just to reconsider the cognitive analogy between human programmers and machine interpreters. They are also efforts to distribute and coordinate cognitive labor across entire software systems, treating human and machine alike as modular, encapsulated components.

Although cybernetic theory is never explicitly invoked by the engineers who wrote about the software crisis, it remains an important point of popular engagement with computing in the 1960s, 70s, and 80s. Whereas engineers writing about software design during this period de-emphasized philosophy and cybernetic theory in favor of concepts that they felt were more objective descriptors of their labor, like "efficiency" or "stability," fiction like Robert Coover's *Universal Baseball Association, J. Henry Waugh, Prop.* (1968) and Margaret Atwood's *The Handmaid's Tale* (1986) actively explore the experience of an awareness fragmented across technological and social structures. Anne Balsamo argues that literature can resist the "uncritical

embrace" of science and technology prevalent in Western culture. If literature and technical writing share a form of speculative reasoning, the primary difference between them, she explains, is that literature allows us to engage in speculation "against the grain" of narratives of scientific progress (114). Elsewhere, Robert Markley argues similarly, calling on literary critics to read scientific papers alongside science fiction because both try to imagine futures made possible through technology, sharing a "fascination with reconceiving the relationship between humankind and technology and, more broadly, between humankind and a complex material reality" (21). Because science fiction is more open-ended, and not subject to the same institutional demands of scientific practice, it can help highlight "the assumptions, values, and beliefs that are taken for granted, half articulated, or defended vigorously as the self-evident parameters of good judgment and common sense" within the technoscientific imagination (12). In addition to reflecting the ways that cybernetics continue to influence the development of computer programming languages, these two novels provide an opportunity to explore the social dimensions of OOP not considered in technical papers, or those power dynamics overlooked by project management textbooks, thereby helping us to explore the cultural narratives underlying technological progress that are often sublimated in technical literature.

Reinterpreting Brian Rotman's cultural analysis of mathematics to account for the differences between calculation and computation can help us to understand how the software crisis disrupted a model of programming based on an analogy between human and machine. In *Ad Infinitum: The Ghost in Turing's Machine* (1993), Rotman explores the implications of a constructivist mathematics, a doctrine which holds that numbers only exist insofar as they can potentially be calculated by known mathematical operations rather than existing as part of a Platonic infinity, awaiting discovery. Throughout its history, Rotman explains, mathematics has

embraced a Platonist ideology by self-consciously constructing a tripartite model of mathematicians: a Person who exists in a realm of imperfect cultural signifiers, an ideal Subject who interprets and applies mathematical laws in an objective fashion, and an Agent that mechanically carries out the operations the Subject forms through combinations of laws. While the Subject may be able to imagine infinity, the Agent must actually produce each number he or she considers. The Agent is thus revealed to be the Subject's mathematical imagination as well as its calculating automaton. The advent of computers, he notes near the end of his study, challenges this model because numbers can only exist within the memory of a mechanical computer through construction. Calculation carried out by computers disembodies the Subject and the Agent from the human mathematician; but mechanical calculation also exposes him or her as embodied by revealing that the machine Subject and its Agent are capable of imagining proofs far beyond the cognitive limits of their human counterparts. The software crisis represents a very similar moment of cybernetic awareness, wherein programmers in the 1960s were forced to acknowledge that the machines they worked with were capable of supporting software systems on a much larger scale than humanly conceivable. This chapter thus begins with a review and elaboration of Rotman's embodied mathematician into a model of embodied programmer in order to draw out the cybernetic elements of the software crisis.

Following a discussion of Rotman, this chapter then applies the model of embodied programmer to Coover's *Universal Baseball Association* in order to show how its cybernetic narrative participates in the discourse of technological collapse and crisis emerging from computing circles in the 1960s. As an accountant, Henry Waugh performed long series of rote mathematical calculations every day. In his free time, he invents a baseball dice game with a set of rules that eventually become so complex he can only carry them out without knowing where

the rules will take him, a thrilling experience that quickly turns disastrous. Henry's crisis following the death of Damien Rutherford represents his inability, or refusal, to understand the decisions that he makes via his mathematical imagination as a programmer. The similarities between Henry and the fears expressed by programmers during the software crisis thus represent a moment in which computers were capable of carrying out calculations beyond the cognitive limitations of the embodied programmer's mathematical imagination. Software, far from being a largely mathematical enterprise, is a complex form of information management. Coover's novel reminds us that software has a complicated legibility that meshes the syntax of mathematics with the signifiers of other cultural fields. If early, informal models of computer science education implicitly assumed a closeness between human and machine, *The Universal Baseball Association* and the software crisis demonstrate that these two forms of embodiment produce distinctly different relationships to software's signifiers.

The second section of this chapter examines how OOP, as a solution to the problems of legibility posed by the software crisis, engenders a form of management that organizes programmers and software in an effort to restore an idealized human-machine cognitive analogy. As Mirowski suggests, while cybernetics developed and expanded from an early branch of computer science into a model for studying complexity in biological systems, the early fascination with human-machine circuits continued through the study of game theory in the field of operations research. Atwood's dystopia in *The Handmaid's Tale* reflects the way that theories of "bounded rationality" informed industrial management practices, and in particular those management styles supported by the changes OOP introduces to the practice of programming. Gilead's requirement that all of its members follow narrowly defined social rules, complete with specific rituals for interpersonal communication, resonates with OOP's practices of modular

encapsulation. If Atwood's dystopia implies some of the cultural consequences of the ways that OOP's organizational style can be exploited to alienate programmers from system design, her novel also suggests that encapsulation doubly binds the agency of programmers by constraining them to only those implementations made possible by the lowest level components in a software platform. OOP, in other words, poses its own problems of legibility through the tensions produced by its emphasis on information hiding.

I. Cybernetics and Software Design

Computer programming appears to fulfill cybernetics' earliest promises. Wiener, for instance, identifies our ability to communicate with machines as a key assumption separating cybernetics from engineering: "We ordinarily think of communication and language as being directed from person to person. However, it is quite possible for a person to talk to a machine, a machine to a person, and a machine to a machine" (76). The sort of communication that humans do with machines, he adds, is different from the interactions they have with other non-humans because machines are capable of encoding/decoding information in ways that resemble human linguistic interactions. In creating machines that can receive, interpret, and respond to encoded communications, we invest machines with qualities "not found among the lower members of the animal community" (77). Programming software, if nothing else, is a continual process of passing encoded language back and forth between human and machine. Both parties constantly translate semantic information into behaviors—chains of commands—and vice versa. Yet Wiener cautions his readers to remember that this form of communication is also subject to entropy, with information lost at each stage of translation. While programming was originally practiced as an extension of mathematics, the software crisis of the 1960s reveals that it is not a purely mathematical exercise; rather, programming is an exercise in information management

that mixes the syntax of mathematics with the signifiers of natural language through its efforts to model sociocultural processes and present information about them to human users.

The history of humans working as computers for mathematicians prior to the 1940s is well documented; however, one of the most detailed accounts of the mental experience of performing calculations comes from Brian Rotman's work on the philosophy of mathematics.[4] Rotman describes an encounter with a mathematical proposition as a tripartite experience. According to Rotman, this structure emerges out of the assumption that mathematical thinking is more important than mathematical writing because the former is believed to be signified-driven and hence free from cultural signifiers that make the latter so imprecise: "mathematics is before all else self-consciously produced. . .out of the image of itself as the exercise and play of pure, abstract reason engaged in the production of indubitable truths" (25). Achieving this pure, abstract reason requires a self-conscious distancing of the self from mathematical reason, a splitting of subjectivity into what Rotman calls the Person, the Subject, and the Agent. The Person is embodied, existing in a culture of signifiers. Although the Person can discuss mathematics, he or she does so via access to an indexical "metaCode" that refers to abstract truth but does so using arbitrary signifiers that are "vague, permeated by illogic and subjectivity, intuitive, uncategorized, immersed in history and cultural mutability" (70). The Subject, on the other hand, is disembodied and exists in a non-arbitrary discourse of mathematical syntax. He or she is able to interpret mathematical "Code" as a form of "pure writing" that is "precise, logical,

[4] For a historical study of human computers see David Alan Grier's *When Computers Were Human* (2005). For a cultural history of the transition from human to mechanical computers, see Wendy Chun's *Programmed Visions: Software and Memory* (2011).

objective, ordered, atemporal, rational, and unchangeable" (70). The Subject reasons through problems by selecting and applying rules drawn out of the Code. Finally, the Agent is a "simulacrum" of the Subject, incapable of its own reasoning or observations but able to carry out the operations selected and arranged by the Subject. Importantly, the Person has no direct access to the Agent because it operates solely at the behest of the Subject. The Agent is thus a ghostly hand that carries out the mental labors of the Subject, serving as the latter's mathematical imagination.

The Subject is able to keep mathematics "pure" by simultaneously shielding the mathematician's awareness from the cultural signifiers of the Person and the algorithmic construction of numbers performed by the Agent. The Subject of classical mathematics, according to Rotman, operates in a realm that emphasizes the signified, tracing and discovering Platonic truths by probing the Code via the Agent. Because the Person communicates through signifiers, he or she has no direct access to this realm. The Agent exists within the realm of signified with the Subject. In practice, however, the Agent is both the Subject's proxy and its ghost. For the Subject to imagine a law, the Agent must be able to produce it; however, for that law to exist as mathematical truth, it must also exist independently of the Agent's labors.[5] In the Platonic model, numbers already exist awaiting discovery. The Agent doesn't calculate them; rather, the Agent moves to their position on an infinite number line by following the directions of the Subject. As a consequence of this model, the history of mathematics is always a narrative of discovery but never one that situates the practice of mathematics within specific cultural moments. Those moments belong to the Person because the Subject exists in direct relation to

[5] For a more detailed account than this summary, see pgs 75-84 in Rotman.

timeless truths. But for critics of the Platonist model, according to Rotman, infinity is a challenging concept to construct. While it may easy to imagine that a number like 5, for example, exists independently of any calculation used to produce it, such as 1+1+1+1+1, it is much harder to deal with a concept like infinity which by definition can never be reached via calculation. For the Platonists, infinity is proof that their view of mathematics is correct: it is mathematical truth represented by laws independent of calculation. Yet for constructivists, infinity is itself a sort of crisis, as it suggests that there elements of mathematics that may beyond human comprehension.

Rotman builds on this idea, positing that all aspects of the tripartite model are embodied and therefore subject to the limitations of human signification and cognition. As a consequence, the Subject not only operates within the same cultural framework of the Person but is also constrained by the limitations of the Person's body. Rotman explains that the Subject, as classically conceived, is "free of error, misreadings, boredom, fatigue, memory loss, and misperception, insofar as these affect the reading/writing and manipulation of Code determined signs" (102). Similarly, the Agent is "an automaton, a wholly mechanical and formal proxy for the Subject" (77). The Agent never fails to execute the Subject's selection of rules, just as the Subject never fails to interpret the Code correctly. But concepts like infinity or pi can be never fully represented by the Agent, and thus understood by the Subject, due to the limitations of the Person's mind. At best, we can only understand them functionally but never completely. Rotman's description of the Agent as an automaton makes it very easy to imagine a digital computer. Rotman, in fact, later posits that the "the advent of computer-aided reasoning in mathematics" has "forced into the open the question of" limitations to the Subject (103). Is a machine, in other words, more capable of accessing the "pure truth" of mathematics than a

human? If a mechanical Agent surpasses the cognitive limits of the human, can a human Subject understand its truth? Does the Subject exist in the human, the machine, or both? While noting that David Hume and Rene Descartes have worried about the way that longer proofs could hinder our ability to follow mathematic reasoning, Rotman, invites his readers to consider how cybernetics could further re-imagine this tripartite model.

There are some key differences between programming and the mathematics Rotman describes. In software design, the Subject is the source code that links human Person and mechanical Agent, existing simultaneously in the human during the act of writing source code and in the machine as the algorithmic processes resulting from that source code, directing the chains of commands executed by machine's hardware, the Agent. The human-aspect of the programming Subject is also less distinct from the Person because most programming languages are a mixture of mathematical notation and natural language. Yet for the machine Subject, source code is always Rotman's Code: algorithms that manipulate discrete information according to very narrowly defined rules with no consideration of the significance of the signifiers chosen by the human to represent those rules. For the human Subject, the source code is inseparable from the metaCode, bound up in the signifiers the Person uses to describe process flow but which are simply just placeholders in the machine Subject's memory: variable names chosen by the programmer, commands represented as natural language words, and non-binary numbers. This blurring of metaCode and Code in programming languages also means that the human Subject is not infallible. The machine Subject will always direct the mechanical Agent's operations exactly as described in the Code, regardless of its size and unaffected by the signifiers chosen by programmers to represent processes. Because the human Subject and machine Subject are differently embodied, they read source code differently. For the machine, source code is pure

syntax; but for the human, source code is a mixture of mathematics and cultural signifiers and thus potentially illegible. Humans can misread source code, find that it is too large to comprehend completely, or simply not chose variable or process names that accurately represent their work to others. Like classical mathematics, the earliest models of programming assumed that humans could filter their awareness, escape the limitations of their bodies, and operate purely in the realm of the Subject. Programmers, in other words, understood the human-machine circuit as an analogy. The more they could think like the Turing machines they designed Code for, the more efficient and more powerful their software would be; however, the software crisis exposes limits to this analogy, raising concerns that the cognitive difference between human and machine is insurmountable as software systems increase in scale.

II. Paper Machines

Although historians of technology have linked the software crisis to other popular visions of technological failure in the 1960s, the software crisis is different from them in that can be defined in distinctly cybernetic terms. Martin Campbell-Kelly and William Aspray claim, for example, that the failure of the Mariner I spacecraft in 1962 due to a single typo in its guidance software served as a dramatic symbol of the fears among computer scientists during the 1960s. In a laboratory setting, this error would have been relatively mundane. Outside of the laboratory, the error resulted in a disastrous spectacle: the mid-flight detonation of a symbol of American power that cost millions of taxpayer dollars. Embellishments of the incident in computer science textbooks and the popular press meant that it "required little imagination to see the terrible consequences if a similar software problem were to occur in a manned space flight, a nuclear power plant, or a fly-by-wire airplane" (200). Yet popular culture during the 1960s was happy to imagine these scenarios. That same year, for instance, Eugene Burdick and Harvey Wheeler

published *Fail-Safe*, a novel that depicts the possibility of nuclear war resulting from a single defective electrical circuit. In both cases, catastrophic failure results from an explicitly defective component. But the concerns raised during the software crisis discuss a failure that is not as easy to locate. Instead, software engineers responding to the crisis worry about systems that fail precisely because they are faithfully executing their software. They worry that software systems have grown so large that humans can no longer comprehend them; as a consequence, these massive systems produce unexpected, potentially disastrous, results even though they continue to perform as instructed. The software crisis, in this sense, resembles the conflict at the heart of Coover's *Universal Baseball Association*, stemming from a problem of legibility, of the inability of programmers to follow the workings of the very systems they defined through source code. While Campbell-Kelly and Aspray's emphasis on the Mariner I oversimplifies the software crisis, it nonetheless exposes a sublimated fear that computer science during the 1960s was growing into something other than an extension of mathematics. As computer systems increased in power, capable of modeling and participating in real-world processes, unexpected results would be more than just an anomaly. The same technologies that were celebrated as leading us towards fantastic futures could also produce devastating failures.

Software engineers, computer science educators, and historians frequently identify IBM's System/360 computer and its OS/360 operating system software as the best representations of the problems programmers struggled with during the 1960s. According to Paul Ceruzzi, the company had invested so much time, manpower, and money into the System/360 that its development struggles "almost sank the company" (101). IBM intended for its System/360 to be the first mass-produced computer system. Yet to be a one-size-fits-all machine, the System/360 needed to be built to handle a variety of use scenarios. The main problem behind System/360's

troubled development was also one of its celebrated features: increased memory and processing power permitted programmers to build more features into the operating system than they were accustomed to managing. Campbell-Kelly and Aspray describe its operating system as "the biggest and most complex program artifact that had ever been attempted [by the 1960s] . . . consist[ing] of hundreds of programming components, totaling more than a million lines of code, all of which had to work in concert without a hiccup" (197). Prior to the 1960s, computer systems were unique, constructed in academic laboratories by electrical engineers or built to order for corporate clients by companies like IBM. The System/360 was different in that it was designed to be flexible, serving needs that could not be anticipated fully at the time of construction. One of the engineers credited with initiating changes to programming education in response to the crisis, Edsger W. Dijkstra, explains in a 1972 lecture that the problems surrounding the System/360 were not surprising: "as the power of available machines increased by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in the exploded field of tension between ends and means" (116). Problems in the OS/360 software became hard to diagnose and were often "corrected" by adding in new processes or completely replacing them in the hopes that new implementations would remove the problematic algorithms.

In cybernetic terms, the System/360 and other projects that struggled during the 1960s represent a limit to the human-machine circuit. Most computer programming languages in use at the time fit into the "procedural" paradigm, modeling software as a process: a linear sequence of commands addressed one at a time. Procedural languages thus resemble Alan Turing's early descriptions of a computer:

> At any moment there is one symbol in the machine; it is called the scanned

> symbol. The machine can alter the scanned symbol and its behaviour is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behaviour of the machine. However the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings (413).

With this model, software is a process that follows a list of commands in sequence, addressing each operation one at a time. Complex procedures can be created using "flow control" operators like the "goto" command that instructs the software to jump to a particular position—moving Turing's tape forward or backward—in the process. Programmers in the 1950s and 60s were usually either self-taught or learned about software design informally from others working in the same laboratory. Computer hardware was not yet mass-produced, and programmers learned to work with specific machines, often in a language that issued instructions directly to hardware.[6] Because software at this time was tailored to specific machines, "programs would [have] only local significance" and very little "lasting value" (Dijkstra 114). From a cybernetic perspective, a model of programming that uses low-level, procedural languages and issues commands directly to specific hardware configurations rewarded programmers who immersed themselves in a human-machine circuit. In this paradigm, the quality of software is a reflection of programmers' ability to think like a Turing machine. Yet the System/360's complexity challenged this model, because programmers found they could not wholly anticipate the effect that individual contributions would have on the system as a whole.

---

[6] This type of programming language is often referred to as an "Assembly" language because it consists of one-to-one instructions used to "assemble" the binary code executed by machines.

Coover's *The Universal Baseball Association* explores in detail a programmer's attempts to think like a Turing machine. The novel tells the story of Henry Waugh, an accountant whose life is intertwined with the dice rolls and look-up tables of a pen and paper game, precursors to the complex simulation models that underlie modern software. While later in his career Coover not only writes hypertext fiction but also heavily incorporates elements of it into his print work, there is evidence that he was already thinking about the relationship between computers and literature in the 1960s. In a 1968 interview, for instance, Coover recalls speaking with an engineer about how early computer scientists modeled their work in hardware and software on "the notion of a human brain as being a central . . . organizing system that sends and receives messages back [ellipses original]" (Hertzel 27). Implicitly, Coover's conversation with the engineer points to the early contributions made by cybernetics to computer science. Yet the engineer also notes that computing has since moved away from this model, finding better ways to design machines that didn't try to model human thought. Implicitly, the engineer points to the limitations of the older procedural style and the need to differentiate between the human mind and Turing machines. Coover speculates that maybe art, too, "compromises our notion of the brain" as a centralized information processor by "reach[ing] the whole body at once" (Hertzel 27). Coover's interest in cybernetics here lies less in the way the human mind influenced computing but in the way interactions with computers have influenced human thought. The *Universal Baseball Association* reflects this conversation in the way that Henry experiences his imaginary baseball league on a visceral level, growing aroused or physically ill as he computes each sequence. His baseball game is an algorithm so complex that it introduces an unexpected emotional dimension. He experiences Damon Rutherford's death not as a statistical phenomenon but as the death of a son. More importantly, Coover's characterization of software in the novel is

one that entangles mathematics with cultural signifiers. Henry's efforts to think like a Turing machine are continually complicated by a desire for his algorithms to do something more than just advance the game's state.

Despite not running on a digital computer, Henry's Universal Baseball Association game (UBA) is a form of software. Modern computer systems are at their core Turing machines: finite state machines that update stored, discrete values according to a specified set of rules.[7] Turing himself explains that "it is possible to produce the effect of a computing machine by writing down a set of rules of procedure and asking a man to carry them out. Such a combination of a man with written instructions will be called a 'Paper Machine.' A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine." (416). As "software," the UBA is composed of the set of rules that procedurally update the state of the paper machine by manipulating stored data. Like any other programmer, Henry assigns unique, human readable names to the system's variables—the league's players—so that he can follow how the system's rules read and write the values assigned to them. As Daniel Punday notes in his history of pen and paper dice games, Henry's UBA structurally emphasizes the unfolding of a game's narrative over its players: "Coover's game may at first seem similar to RPGs" like Dungeons and Dragons "because it relies heavily on statistics" of individual players, interpreted through "complex charts[; however,] these statistics describe not objects, but events" (118). The

[7] For a full definition, see Alan Turing's "On Computable Numbers, with an Application to the Entscheidungs Problem," reprinted in *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma* (2004), ed. B. Jack Copeland.

UBA thus resembles a program written in the procedural paradigm. Formally, the game's algorithm is the continuous iteration of a random number generator: a set of three dice. Each generation is followed by a selection of state change rules determined by the combination of numbers produced by the dice. No matter what phase—play, scheduling, player promotion/demotion, postseason evaluation—the system's state is always advanced in this manner. Subroutines are called when selection criteria are met, but the program is centralized around a single process to which it continually returns. Importantly, the UBA's reflection of procedural principles also allows the novel to illustrate the defining problem of that paradigm: the larger and more complex a procedure-oriented program becomes, the more entangled it becomes with cultural signifiers, making it harder for human observers to trace sequences in its mathematical syntax.

Descriptions of the software crisis, both in accounts from programmers and in Coover's fictionalized narrative, represent the crisis as a moment when the cognitive differences between human and machine could no longer be ignored and the analogy between human and machine Subjects could no longer hold. Unlike humans, computers operate by holding the entirety of software's code in an instantly accessible form of temporary memory during runtime, able instantly to recall any element of the system's state. The machine Subject must be able to imagine every action of the Agent permitted by source code if a computer system is said to be functioning correctly. Yet humans, by contrast, read source code in much the same way they would read other texts, following the flow of signifiers across the space of the page or screen. The human Subject can thus try to imagine the operations of the machine Agent as the Person traces processes by following an algorithm's textual inscription in source code; however, humans do not hold the entirety of a software system in memory as machines do. Human readers can lose

thus track of how or why the system reached a certain state as they cognitively juggle their memory of the machine's state with their navigation of the source code's signifiers. Reading this model back into the software crisis, we can say that even though the Codes of the System/360 and Henry's UBA are formally sound—executable by the machine Subject—human Subjects struggle to imagine their operations because the machine Agent carries them out on a scale and at a speed that human Agents cannot match. That is not say that computer hardware exceeds the computational power of the human mind; rather, the human mind is embodied differently from a Turing machine, unable to match digital hardware in carrying out a particular mode of information processing.[8]

The 1968 NATO Conference in Garmisch, Germany aimed to address this problem of embodied programming, albeit in primarily industrial terms. As editors of the conference report note, they specifically chose to frame the conference around the term "software engineering" in order to foreground "the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering" (13). While the conference's papers are no longer available, the conference report does include quotations from papers and transcriptions from the question and answer sessions

[8] Non-human instances of this problem are well known. For example, older hardware can be "emulated" by software on new machines. Twentieth century video games have enjoyed a second life on smart phones via software that mimics the storage and execution of older cartridge-based consoles. Emulation is very processor intensive, and new computing devices that are ordinarily much faster can struggle to emulate older hardware at the same speed as the original, physical devices.

that explicitly call for an end to informal programming education and systems of rigorous management in an effort to prevent future projects from suffering the same problems as the System/360. For example, A. d'Agapeyeff states explicitly that "[p]rogramming is still too much of an artistic endeavor," arguing that it is no longer possible for programmers to work in relative isolation. (24). In order to avoid the real possibility of changes to one component derailing others, programmers must instead foreground principles of feedback between components as mechanical and electrical engineering already did (23-24). Comments from Garmisch attendees reflect the assessment of the System/360's chief project manager, Fred Brooks. Writing in 1975, Brooks argues that while "two programmers in a remodeled garage [could build] an important program that surpasses the best efforts of large teams," such projects were no longer feasible because mass production meant that it was impossible to predict every single state a software system might reach (8-9). In other words, programmers needed to focus on stability rather than fixate on ideals of mathematical efficiency. Beyond producing machines for use in large corporations, many in attendance also recognized that computers were now a part of social infrastructure. Peter Naur, for example, comments that because computers are affecting more and more people directly, "software designers are [now] in a similar position to architects and civil engineers. . . .It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem" (35). Programmers, in other words, had to address the fact that software's entanglement of human culture and mathematical syntax meant that illegibility was now a pervasive structural or design problem that reached beyond small laboratory communities.

The Garmisch attendees implicitly acknowledged the gap between the human and machine Subjects in their assertion that the problems of the software crisis are inevitable within the procedural paradigm. A. Opler argues that the problem of software legibility will only get

worse as the "the current growth of systems" produces "what [he] expect[s] is probably an *exponential growth of errors* [emphasis added]" (17). Failures like the System/360, he implies, will become more frequent unless programmers find a way to account for the distinct ways that human and machine embodiment shapes the mathematical imagination of both. Even in the case of attendees such as Ken Kolence, who viewed talk of a "crisis" as alarmist, there is nonetheless an acknowledgement that programming as then practiced was incapable of accounting for the problems of legibility posed by increasingly large software projects: "The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time" (17). Regardless of whether attendees agreed that their present moment should be called a crisis, there is a shared sense that the recent failures highlighted a disconnect between the mathematical imagination of human and machine Subjects. Elsewhere in the report, E. E. David, Jr. and A.G. Fraser explain that what they find "alarming is the *seemingly unavoidable fallibility of large software*, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well [emphasis added]" (16). While cybernetics is never mentioned, the idea that humans cannot think like Turing machines at scale is nonetheless implicit throughout the conference. Understanding how this gap affected the experience of programming requires a closer consideration of accounts of human-computer interaction like the kind found in Coover's novel.

Before returning to Coover, however, it's important to note that problems of legibility were also exacerbated by the informal instruction and localized expertise of programming prior to the 1970s. Joseph Weizenbaum's archetype of the "compulsive programmer" is drawn out of the concerns about pre-software crisis programming education. As Weizenbaum notes,

compulsive programmers are interested in computation as an end in itself. They not are necessarily "bad" at programming, but their work is unstructured, guided primarily by a desire to test the limits of systems and demonstrate their own creative genius. Unsurprisingly, Weizenbaum's model is often cited as the first representation of people who are today referred to as "hackers." Compulsive programmers, he notes, design software "without plan and without knowledge, let alone understanding, of the deeper structural issues involved," dividing their time between "piling new subsystems onto the structure [they have] already built. . . [and] attempt[ing] to account for the way in which substructures already in place misbehave" (119). Although not all programmers working in the procedural paradigm were compulsive, low-level flow controls like the goto command permit an unchecked increase in complexity. Compulsive programmers could simply splice new subsystems into large processes ad hoc by re-directing the process flow. So long as they were careful to return the program back to the right place in main process flow after a new subsystem was executed, this practice would not, in theory, disrupt the program. The machine Subject and its Agent have no trouble interpreting these complex process flows; yet the human Subject, reading through a mathematical syntax entangled with cultural signifiers can have trouble navigating what many referred to as "spaghetti code." Like the classical mathematicians Rotman describes, compulsive programmers have little regard for human readers of their source code. They value their identity as Subjects, trying to think primarily at the level of Code and suppressing those aspects of source code relevant to Persons in order to imagine themselves more intimately connected to the pure computation of the machine.

Henry very much reflects the behavior of the informally trained, compulsive programmer. As Coover explains, in the past Henry drifted from game to game, continually seeking new challenges: "He'd always played a lot games: baseball, basketball, different card

games, war and finance games, horseracing, football, and so on, all on paper of course" (44). Before the Universal Baseball Association, he had very little interest in any particular game but was instead interested in exploring new computational systems. In addition to writing games, he would also modify existing ones by adding new rules systems and mechanic components. He once created

> a variation on Monopoly, using twelve, sixteen, or twenty-four boards at once and an unlimited number of players, which opened up the possibility of wars run by industrial giants with investments on several boards at once, the buying off of whole governments, the emergence of international communications and utilities barons, strikes and rebellions by the slumdwellers between "Go" and "Jail," revolutionary subversion and sabotage with sympathetic ties across the boards, [and] the creation of international regulatory bodies by the established power cliques (44)

As a compulsive programmer, Henry thrives on his ability to splice new subprocesses into game. He is able to follow these complex procedures because he is able to alienate himself from his identity as a Person. He represents a fictional embodiment of the procedural paradigm, able to think like a Turing Machine by enacting a cognitive separation between his Person and Subject. Yet in expanding the Code of his games, Henry is also increasingly entangling the rules of the UBA with signifiers drawn from outside mathematics. He continually invents new rules modeled after the sport, splicing them into the game between seasons in order to expand the game's potential to produce narratives of the UBA's great players. In doing so, Henry begins to lose track of the rules. He even admits to himself that, by the opening of the novel, the UBA was "slowly buckling under a kind of long-run market vulnerability, the kind that had killed off

complex games of his in the past" (136). Although there are routine algorithms that he remembers easily, the sudden death of Damon Rutherford is surprising because the UBA's system of algorithms has become illegible. Henry can carry out its rules by consulting the tables of Code that he designed, but he cannot remember all of the system states he defined in them.

In this respect, Henry's UBA reflects the ways that the procedural paradigm privileges machine over human legibility. One of the first calls for reform among programmers came in Dijkstra's "GOTO Statement Considered Harmful," a letter published in the March 1968 issue of *Communications of the ACM*. In it, Dijkstra argues that an over-reliance on the goto command discouraged careful planning because programmers could simply redirect the main process of a piece of software at will. While goto is no longer found in programming languages developed during or after the 1980s, it was a staple of the procedural paradigm. Programmers could create loops and develop subprocesses by using conditional statements that moved the process forward or backward to specific numbered lines in a large body of textual commands. Yet Dijkstra observed that relying on goto for flow control created a gap "between the static program and the dynamic process" (147). From the perspective of the machine, the Subject always directs the Agent to proceed from one command to the next. Even if the Code directs the Subject backwards to a command that the Agent previously had executed or passed over, the Subject and Agent experience it simply as the next command in a sequence. Because the machine holds the entire program sequence in memory and can direct the Agent instantly to each line specified by goto statements, the placement of subprocesses is arbitrary. But humans must read the sequence within the source code much as they would a natural language text: from left to right, top to bottom. Flow controls like goto commands, Dijkstra argues, interrupt this sequence and force the human Subjects to wait on embodied Persons to halt their normal reading process and locate the

next line they are directed toward. The goto command, in other words, served as a key representation in the way embodiment shaped software's legibility. Dijkstra's letter proved controversial, drawing defensive responses for nearly 20 years from programmers unwilling to cede control of their processes to automated flow control operators—like "while" and "for" loops.[9] Rather than acknowledge Dijkstra's point about the differences between human and machine cognition, many of the letter's critics interpreted the argument against goto statements as one about efficiency, offering computational proofs to show that implementing automated flow control statements required more variables or lines of code.[10] These responses indicate that many programmers still idealized the model of programming as thinking like a Turing machine. They claim to sidestep the problems of legibility posed by the System/360 and the UBA by accusing Dijkstra and his supporters of having a flawed mathematical imagination without recognizing that it is precisely this idea of a "flawed," embodied subjectivity that Dijkstra tried to address.

Dijkstra further developed his argument against the use of the goto command into "structured programming," a set of stylistic recommendations for that procedural paradigm that later became incorporated into most OOP languages. In 1972, Dijkstra published his "Notes on

---

[9] The most recent response is "'Goto Considered Harmful' Considered Harmful" by Frank Rubin, *Communications of the ACM*, March 1987.

[10] Most of the initial responses in subsequent issues of *Communications of the ACM* are little more than short proofs, but Donald Knuth and R.W. Floyd's "Notes on Avoiding 'Go To' Statements," in *Information Processing Letters* (1971) is by far the longest and most detailed response.

Structured Programming," a long-form essay that describes methodological solutions to the "widespread underestimation of the specific difficulties of size [that] seems [to be] one of the major underlying causes of. . .software failure" (2). Structured programming takes legibility as a primary goal of design, acknowledging that programming is not a purely mathematical exercise governed primarily by the Subject but relies heavily on the Person, especially as software projects grow from single-author texts into large, collaborative endeavors. Rather than privileging the idea of a mathematical brilliance that casts programming as "primarily the minimization of a cost/performance ratio," Dijkstra calls on programmers to acknowledge their limitations as readers of source code: "we can only afford to optimize (whatever that may be) provided the program remains sufficiently manageable" (6). While much of the essay talks through informal mathematical proofs supporting his reasoning, Dijkstra importantly proposes the importance of the "textual index" in source code composition. Revisions to languages like Pascal that incorporated structural principles in practice encouraged chunking code in long, complex programs so that subprocesses would be visibly localized to particular segments of the source code text. Although data patterns in large systems may be difficult to predict and may still reach a point beyond human comprehension, Dijkstra's essay implies, the names and visual organization of variables and subprocesses are something programmers can control and should be able to interpret readily. Readers of source code should be able quickly and readily to identify how a machine will progress through a set of instructions even if they cannot imagine in advance every single calculation sequence that might be performed by the machine Agent.

Having no audience other than himself, Henry and his UBA take the problem of source code legibility to an extreme. Although Henry develops emotional attachments to elements of the game defined by his Person, they are based on the results of the machine Subject's algorithms. In

choosing to develop a baseball game, it was "[n]ot the actual game so much—to tell the truth, real baseball bored him"—that drove his development of the UBA's rules, but rather "the accountability—the beauty of the records system which found a place to keep forever each least action—that had led Henry to baseball as his final great project" (45; 19). Henry is only attracted to the computational side of the game: "the records, the statistics, the peculiar balances between individual and team, offense and defense, strategy and luck, accident and pattern, power and intelligence" (45). Even though Henry claims that the "dice and charts and other paraphernalia [are] only the mechanics of the drama, not the drama itself," his anecdotes show otherwise (47). For instance, the meaning and importance of the names he assigns to players are derived from those mechanics. Despite remarking to himself that a player he names "Old Fennimore McCaffree" [is] defined by something outside of the games rules—"that [is] simply who he [is] . . .Scholar and statesman. Dark. Angular. Intense"—Henry admits that Old Fennimore wouldn't have been one of the oldest rookies in the league if it were not for "an unlucky throw of the dice on the Rookie Age Chart" (47). In other cases, such as the events around Brock Rutherford's sons that drive the novel's central conflict, family names are arbitrarily chosen, and the dice later determine the significance of their paternal relationships. The league history that Henry meticulously describes in his journals is always at a minimum a record of statistics—a report of the Agent's activities generated by the Subject—often but not always embellished with narrative flourishes. The game's algorithms do not exist to recreate baseball; rather, baseball exists as a way for Henry to remember its statistical history. Even though he performs his Subject identity like a Turing machine, he is still embodied as a human. He cannot hold the entire system's mathematical state in his head and therefore requires lists and tables that he can search through to aid his memory.

For compulsive programmers, there is always a hope that they'll discover in tracing the Agent's operations rules that were applied by the machine Subject in ways they had not anticipated. Weizenbaum notes that compulsive programmers were thrilled to see computer systems "misbehave in a number of mysterious, apparently unrelated ways" because it "gives every evidence of [the system] having taken on a life of its own, and certainly, of having slipped from [their] control" (119). While a machine behaving seemingly on its own challenges the human-machine analogy, it also represents an opportunity to grow even closer, revealing an as yet unknown aspect of the machine Subject. If machine Code that executes is never wrong, then the unexpected for the compulsive programmer is simply some part of the machine that the human Subject does not fully understand. The unparalleled success of a rookie player Henry arbitrary names "Damon Rutherford," for instance, gives Henry the sense that his game is more than he imagined it to be, instilling a belief that the UBA's Code contains some deeper truth beyond mere baseball statistics. By assigning the player the last name "Rutherford," Henry imagines him to be the son of a past player, Brock Rutherford, who had set a number of league records but may very well pale in comparison to his son. Damon, though, was not the first time Henry had imagined a rookie to be Brock's son; however, the first had not generated comparable statistics. The UBA's machine Subject had previously sent "Brock, Jr." into an early retirement. To see Damon early in his first year accrue results resembling those of his father "had been a wonderful league tonic" (104).

The problem with Henry's view of Damon's success, however, is that it is a result of the game's illegibility. There is little functional distinction, from the perspective of the machine Subject, in the rolls that made Damon famous and those that led to his death. From the perspective of the machine Subject, "Damon Rutherford" is just a label for a set of statistics

produced by the Agent. Henry's desire, as a Person, to see Brock's legacy continue constrains his embodied perspective as Subject. He focuses only on the statistical anomaly of Damon's sudden success without considering that such a combination of dice just as easily could lead to catastrophe. He admits that past players who enjoyed rapid success seemed to burn out after their first year, but he refuses to believe Damon could meet such a fate. During the same game that Henry begins to identify with Damon as his own son, the dice call for him to be fatally struck by a line drive. Within Henry's system, the scenario that leads to Damon's death is the result of rolling three one's after the system state called for him to consult the Extraordinary Circumstances table. The table was one Henry "still hadn't memorized. For one thing," he admits, "it didn't get used much, seldom more than once a season; for another, it was pretty complicated" (69). The chart exists to invest the game with a sense of drama beyond the simple mechanics of baseball. Of all the charts in the game, it is the clearest example of the signifiers of the Person entangling the syntax of the Subject. There are no statistical representations that readily capture the events listed on the table: "[r]ain could end the game, a drunken fan could crack a player's skull with a pitched beer bottle, a brawl could break out, game-throwing scandals could be discovered, [or] epidemics of dysentery could ravage a line-up" (70). If no other player has died under these circumstances, it is only because that combination of three one's had never been invoked when the machine Subject consulted the chart in the middle of a pitching state. Although Henry wrote the rule, he likely never imagined his machine Subject executing it on a player like Damon Rutherford. The system has become so complex, so illegible, that he never considered the circumstances under which the "Batter struck fatally by bean ball" rule would be invoked when he added it to the game's Code. Importantly, Coover reminds us, it only takes a single instance of illegibility, a single possibility not considered, for a software

system to produce a catastrophic failure. The engineers at Garmisch didn't fear a single typo like the one that resulted in the Mariner I destruction—that was something they believed they could control. Instead, they feared the limits of their own mathematical imagination, and the consequences of machines that could operate beyond those limits.

III. Freedom From Complexity

Solutions to the software crisis coming out of Garmisch emphasized minimizing the amount of information programmers had to manage when writing software. Software was growing in size and now required large teams that had to be coordinated in ways that avoided the problem of legibility by minimizing the opportunity for misinterpretation of system processes. Despite several in attendance at the conference calling for broad interpersonal communication between project members, those responses arguing for its constraint reflect the way that "encapsulation" would reshape programming by the 1980s. Dr. J. Nash, for example, proposed that there "are dangers in uncontrolled mass-communication. You can get into trouble if people start taking advantage of information that they gain by chatting that they should not know (and which may well lose its validity in a day or so)" (90). Similarly, David comments that project managers must "avoid flooding people with so much information that they ignore it all. Selective dissemination of information. . .should be tried in a large software project" (91). Modular encapsulation, a core principle of OOP, calls for fragmenting software systems in ways that restrict information flow between components. By dividing the machine Subject into a hierarchy of distinct components, OOP effectively narrows the mathematical imagination of each component's Subject so that it remains closer in scale to that of the human Subject. Programmers are no longer responsible for trying to imagine the effect of their algorithms on the whole system; instead, they only have to address all of the possible states of a single component. As

long as each component passes the information hidden within it to others upon request, the entire system remains stable. As a response to the software crisis, OOP promised that programmers could make changes within components without worrying about how those changes might result in unimagined states elsewhere.

But as Nash and David imply, OOP changed more than the individual practice of programming. System design became as much about managing relationships between programmers as it did about complex networks of technological components. Among the Garmisch attendees, Dijkstra is the only one who draws this parallel explicitly. He proposes that "both the total density of information flow necessary between groups, and the percentage of irrelevant information that a given group gets, can be greatly reduced by effectively structuring the object to be constructed and ensuring that this structure is reflected in the structure of the organization making the product" (89). Because OOP permits programmers to work on components in relative isolation from one another, component structures become mapped onto social structures. Today, technical papers and reference manuals explaining OOP languages do not acknowledge this aspect of the paradigm. Only textbooks and articles on engineering management explore organizing teams into modules, controlling and evaluating the coordination between them, and methods of outsourcing component development.[11]

Fortunately, the extra-technical consequences of modularization have not escaped the attention of digital media studies; however, the question of how OOP affects the cultural study of software remains open. Tara McPherson argues in "U.S. Operating Systems at Mid-Century: The

---

[11] See for example *Project Management* (1999), ed. Paul Trinello and *Management of the Object-Oriented Development Process* (2006) by Liping Liu and Boris Roussev.

Intertwining of Race and UNIX," that the structural logics driving technological production are not limited to technical or industrial contexts. In particular, McPherson notes that while modularity is celebrated among technologists, it is often oppressive in a social context, functioning as "a mode of partitioning that turned away from the broader forms of alliance-based and globally-inflected political practice that characterized both labor politics and anti-racist organizing in the 1930s and 1940s" (30). McPherson's work highlights that even though technical reasoning is often productive in a limited context, it can be repressive in a broader, social context. Atwood's *The Handmaid's Tale* explores this idea in depth, imagining the consequences of a society structured primarily around modular systems of biological reproduction. The novel not only resonates with OOP's modular encapsulation but it also reflects principles from game theory that underlie industrial management practices, in particular Herbert Simon's "bounded rationality." *The Handmaid's Tale* therefore can help to illuminate the assumptions OOP makes about its programmers and how those assumptions produce their own problems of legibility.

Before I move on to Atwood, however, I want to emphasize that the type of modular organization that OOP permits is not simply an extension of Fordist or Taylorist labor management. As Philip Mirowski's study of how cybernetics influenced economics and corporate management strategies suggests, modern software engineering has much in common with operations research, and not only because the RAND Corporation, a private military think tank, served as an early nexus in the United States for both. Operations research, Mirowski notes, emerged during World War II as a way for mathematicians and scientists to provide expert analysis during military operations—and benefit from generous government funding—without inserting themselves into the military hierarchy. Following the war, operations research in the

United States continued at RAND, where researchers explored a “newfound scientific approach to government, corporate management, and the very conceptualization of society as a cybernetic entity” (182). This last aspect of operations research manifested itself as “game theory,” or the study of human behavior through computational automata. Incidentally, game theory was pioneered by John von Neumann, the same mathematician who designed an implementation of the Turing machine that serves as the model for most modern computers. While Mirowski notes that von Neumann had by 1950 grown skeptical of the field because he no longer believed that Turing machines were a productive analogy for the human mind, one of his colleagues at RAND, Herbert Simon, saw value in the differences between brains and computers. It’s through Simon’s recognition that Turing machines processed information differently from human minds that connections between game theory and programming become visible.

Near the end of the 1950s, Simon’s theory of “bounded rationality” became, at least for a time, a core principle in game theory. Mirowski argues that even though Simon’s role in cybernetics is often downplayed, he is one of the true inheritors of Wiener’s cybernetics. Simon’s bounded rationality posits that “formal logic does not empirically describe how humans think,” but rather that humans reason within narrowly defined circumstances (462). Rather than reach a solution by “grasp[ing] the structure of explanation through mathematical intuition or axiomatic display,” humans often reason through mimicry, shaping their actions based on similarities to prior experiences and information that they draw out of their immediate environment (463). Trying to simulate this model of rationality, Simon produced a number of simulations involving a series of self-contained, problem solving modules. His software produced results that were “locally impressive to the untutored layperson without actually having to solve many of the knottiest problems of the nature and operation of generalized intelligence”

(465). Responding to his critics, Simon argued that intelligence can exist in organizations, embedded in structures that coordinate localized human behavior.[12] Simon, in other words, was interested in considering the possibility that intelligence can exist in feedback circuits between agents. While game theory was shunned by mathematics and scientists for being too "applied," Mirowski argues that it survived by assimilating itself into business and engineering schools as a theory of industrial management. These models still tried to optimize production, but they did so under the assumption that human agents needed systems that guided them towards optimal behaviors. It wasn't until the 1980s, when game theory was rediscovered by economists as part of a "cybernetic wave," that it gained wider attention once more. Importantly, this re-emergence of game theory into the popular imagination coincides with the way industrial management styles were integrated into software engineering by the 1980s. In this respect, OOP becomes a form of organizational intelligence, a way of expanding the powers of the human Subject by constraining both it and the machine Subject within particular segments of the Code. By examining the system of component relationships, software designers can adopt an abstract view of the system, allowing their Person to consider the sum of the system's machine Subjects without being overwhelmed by their collective complexity.

Atwood's novel reconfigures the concept of a networked "bounded rationality" into a community based on the principle of "freedom from" complex social conflicts. *The Handmaid's*

[12] The mathematical logic behind Simon's theories can be found in the following essays: "A Behavioral Model of Rational Choice," *Quarterly Journal of Economics* 69 (1955): 99-118; "Rational Choice and the Structure of Environment," *Psychological Review* 63 (1956): 129-38; and "The Organization of Complex Systems," *Hierarchy Theory*, ed. Howard Patee (1973).

Tale is set in Gilead, a theocratic dystopia located in the United States that forms following a religious civil war. Infertility is now widespread, and those few women who remain able to bear children are classified as Handmaids and pressed into service as surrogate wombs for high-ranking military families. Gilead's social structure rests on the assumption that constraining the decision making of a system's individual participants is the only way to achieve stability in a society constantly under threat of environmental collapse and heretical intrusion. Reflecting on her indoctrination into the ranks of the Handmaids, Offred recalls Aunt Lydia's assertion that there is an important difference between a "freedom to," which defined the "anarchy" of pre-Gilead America, and a "freedom from," which defines the present day (24). Offred reflects on all of the personal responsibilities she had to look after during the time before Gilead: "the rules that were never spelled out but that every woman knew":

> Don't open your door to a stranger, even if he says he is the police. Make him slide his ID under the door. Don't stop on the road to help a motorist pretending to be in trouble. Keep the locks on and keep going. If anyone whistles, don't turn to look. Don't go into a laundromat, by yourself, at night. (24)

Atwood's satire of feminist rhetoric through the lessons of the Aunts is memorable because it reframes a return to women as property as a solution to concerns about women's safety. Even Offred admits that her new status as an obstetric instrument affords her a protection from sexual harassment and the lingering threat of rape she remembers from before the war. Now, she "walk[s] along the same street" with other Handmaids "in red pairs, and no one shouts obscenities at us, speaks to us, touches us. No one whistles" (24). Even if the men she sees privately view her as a sexual object, they cannot act on those thoughts. Like Offred, their private thoughts are bound by the scripts of expected behaviors that structure Gilead's social

system. Atwood's Gilead thus shares the strategies OOP uses to avert the problem of human-machine disanalogy fretted over by engineers during the software crisis. Gilead effectively ends many of the complexities of pre-Gilead gender by restricting freedom of interaction.

In this respect, Atwood's novel stands out from other dystopian narratives because it has a decidedly cybernetic dimension. Control in Gilead exists in the circuits between people, in the rules that dictate the behavior between actors. Comparing Atwood's novel to George Orwell's *1984*, Lois Feuer explains that both dystopias leave their characters to struggle against an ever-present threat of identity loss. Like Orwell's Party systematically re-writing history, Atwood's Gilead submerges individual identity beneath color-coded uniforms and, for Handmaids, new names that sever them from their past lives. Unlike Orwell who feared that the strength of human spirit is "relative and subject to the violence-enforced will of whoever is in power," Feuer argues, "Atwood's point is that the truth of human individuality and (only through this individuality) human connectedness is absolute, inviolable" (88). Read from another perspective, the preservation of individuality loses some of the positive aspects that Feuer attributes to the novel because Offred's personal strength during the course of the narrative becomes reconfigured as a source of torment. Refusing to let go of her pre-Gilead memories, she must continually suppress them and any desire she has to interact with people in ways she had before the coup in order to fulfill her role as a Handmaid. Should she behave unexpectedly, she will be declared an "unwoman" and sent into the wastes. Offred's few interactions outside the household often follow a script of expected greetings and responses. She observes that her clothing, in particular, encapsulates her within her role as a Handmaid. While dressing, she reflects on how her uniform serves as a social interface: her identity defined by its "color of blood" and the white wings around her face which "keep [them] from seeing but also from being seen" (8). Offred's

submission to the state does not require an ultimate act of emotional violence, like Winston's box of rats in *1984*, that forces her to proclaim a lie as truth. Rather, her torture exists in the very framework of Gilead itself and the way it forces her continually to deny her pre-Gilead identity. Resistance in Gilead is a game of prisoner's dilemma, in which two people must step outside of their assigned roles at the same time or risk being reported by the other as defective. Yet because everyone is also encapsulated by his or her role, the decision to resist can only be made internally and not based on knowledge of another person's private identity. Offred's ability to make decisions within Gilead's social system, in other words, is always already bound up with her status as a Handmaid.

Like Gilead, OOP's emphasis on encapsulated modularity also reflects the cybernetic aspects of bound rationality. OOP doesn't do away with the complexity that programmers struggled with in the procedural paradigm. Rather, OOP wraps that complexity in an organizational style that minimizes its potential for entropy. Instead of trying to follow every single operation in a large, tangled process, OOP affords simpler perspectives. Rather than managing a tangle of subprocesses, each manipulating a shared data pool, programmers define independent modules that contain their own data and subprocesses, hiding them from others and only sharing data upon request under specific conditions. Encapsulation, Kay explains, produces "a recursion on the notion of computer itself. Instead of dividing 'computer stuff' into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each. . . object is a recursion on the entire possibilities of the computer."[13] Thus, OOP preserves the close relationship between human and

[13] See: http://www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Introduction.html

machine Subject by binding the latter within an encapsulated framework, shrinking its scale to align more closely with the limitations that the embodied Person places on the mathematical imagination of the human Subject. On the surface, OOP appears to restore computer programming at a local level to the pre-software crisis ideal of a human-machine analogy. While writing the machine Subject's Code for a component, the programmer's Subject is screened off from more complex concerns. Source code is still composed in the signifiers of the Person, but the scope of those signifiers is now encapsulated within a particular component, binding them with a narrow context and decreasing the opportunities for misinterpretation.

The cumulative effect of encapsulation also makes it possible for a programmer to approach the system from an abstract perspective, defining relationships between components without needing to understand the particulars of their internal processes and data. According to Grady Booch (2007), OOP system design is often cyclical. Ideally, writing OOP software begins with an abstract plan of a system that provides a general description of the tasks it should accomplish and of the circumstances under which those tasks might interact with one another. Each task is then defined as concretely and as separately as possible. Many of these tasks will themselves prove to be too abstract and need themselves to be decomposed (14-20). This process continues, until the set of objects defined through each step of decomposition reach a "primitive" state and represent objects or data types that cannot or do not need to be decomposed further. To draw once more on Rotman's terminology, software designers must first describe the system in the abstract, from the perspective of the Person and in the metaCode. With an abstract design in place, they can then settle into the perspective of the human Subject, confining their Code writing activity within modularized sections of the machine Subject. This abstract perspective works in reverse from Rotman's model of the classical mathematician, privileging the metaCode

as a more accurate view of the system than the Code.

If intelligence exists in the relationships between components, then the significance of each module is determined by those components that are more abstract than it. Consider, for example, a word processor. A module must exist to manage the textual signifiers rendered on screen. This signifier module could be decomposed into at least two modules: one that checks for the system's language and loads into a memory a numeric representation of all latters associated with that language's alphabet and another that holds instructions for translating numeric representations into the graphemes of the active font. The signifier module would thus instruct the alphabet module to pass information into the grapheme module and then render the graphemes according to the specifications it receives from the latter. Neither of these more primitive modules have much significance in the system by themselves; instead, they gain purpose through the relationship that the more abstract signifier module produces between them. The signifier module then itself gains purpose through relationships established by a more abstract module defining documents in the word processor's system. Without an abstract module to pass document text to it, the signifier module serves little purpose. Encapsulation is visible in this example in the way that more abstract modules are independently defined. Programmers can view a slice of the system, seeing relationships between modules and the patterns of information that flow between them, but they are blind to the specific operations each module performs on as information passes through them. Nor can they see whether those modules themselves contain their own networks of more primitive modules. Programmers value the hierarchies produced through encapsulation because, unlike the spaghetti code of the procedural paradigm, they can trace readily the flow of information between distinct components, identifying the moment at which information is passed between them in an unexpected form.

Atwood's novel helps expose social tensions that can arise from this organizational framework that are often unacknowledged by technologists. For most members of Gilead, only oral communication is permitted. This restriction reifies Gilead's encapsulation by making sure that most communication is local, one-to-one between its components. Yet for high-ranking male officers, like Commander Fred, there remains some access to written information. Mario Klarer explains that Atwood's dystopia shows how "orality can be deliberately cultivated to cement political structures and for the purpose of eliminating the destabilizing potential inherent" in access to print (130). For Klarer, forced orality is oppressive because "knowledge handed down orally eliminates obsolete words and meanings" effectively "narrow[ing] the perspective, which in turn makes experience something permanently present, preventing historical thinking. All these factors, in their totality, are supposed to make female criticism of a system impossible, although this logic is better described as a way of keeping women's faculty of abstract thought at bay" (133; 136). Reading the cybernetic aspects of OOP in the context of Klarer's comments, we can understand bounded rationality as an access constraint on signifiers, specifically on sites of interpretation and methods of transmission. Commander Fred's rationality is less bound precisely because he has access to a wider range of signifiers than Offred. When Offred asks him why he has access to non-religious materials like old magazines, he replies that Gilead's leaders are permitted to have them because their motives are "beyond reproach" (158). The larger religious mission of Gilead belongs to the Commanders to interpret and is beyond the scope of a Handmaid, a Wife, a Martha, or lower ranking officers and conscripted men of the military whose perspectives are bound to fulfilling their respective charges. From their position as the most abstract components in Gilead, male Commanders can use their access to a broader range of signifiers to determine the best configuration of other people, re-arranging them so that they

can better contribute to Gilead's abstract goal of repopulating the world with good Christians.

While it may be the case in smaller software projects that a single programmer filters his or her perspective between the bound Subject of the component and the abstract Person of the system, it also possible for designers who are responsible for high level system planning to have little or no direct contact with the programmers responsible for implementing individual components. Considered as a labor problem, OOP makes it easy to exploit techniques of encapsulation to draw programmers into projects purely for their intellectual labor without allowing them to contribute to or voice concerns about the project as a whole. The term "code monkey," used playfully by programmers as a reminder that their primary concern is in producing source code, points to this divide between system designers and project staff. As in Gilead, OOP's split perspective can be exploited by forcing portions of a development team to limit their input within the confines of particular components. Project members with a more abstract view of the system are in a position to determine the value and meaning of other members' labor, while those with a more primitive view of the system become alienated from the context of their production. On the surface, this social structure is hierarchical; however, OOP's tendency to re-use encapsulated components also places some power in the lowest levels of its system. Software is rarely written from scratch. Beneath the typical software application lie entire libraries of components that support it. As a result of encapsulation, these lowest levels remain inaccessible, as software applications make requests—like "draw a window for me," "pass my document into the printer spool," or "let me access files on the USB drive"—but have little control over how those requests are fulfilled. These lowest level components, collectively referred to as the "platform," exert their own form of control. In order to remain compatible with a platform's standards, software development projects often incorporate these libraries as a

matter of convenience both in reducing labor costs and in making matters easier for their users, who are presumed to have them installed already. The "code monkeys" are in this respect doubly bound, finding the mathematical imagination of their Code writing Subjects constrained both by the abstract expectations communicated to them from corporate managers and from the rules already in place from companies like Microsoft and Apple, who control the low-level libraries of commands they have to use to build the components they are assigned to by their managers.[14]

Gilead does not escape this tension either. While most critics focus on the hierarchy formed along gender lines through abstraction, Gilead is itself bound by the changes to human biology that occurred following The United States' second civil war. With the structure of Gilead in mind, Atwood's novel demonstrates the limitations placed on individual perspective within an OOP model in two ways. Gilead is ostensibly a theocracy, but the designs of the Commanders are ultimately bound by the limitations on human reproduction. The roles they define within Gilead and the way they assign them within their own homes are always already influenced by the need to overcome widespread infertility. Because the primary task of each household is to repopulate society, the Commanders manage the relationship of their household to the rest of the society, requesting replacements as necessary and passing new members along to be managed directly by their wives. The wife carries out the household's interests locally through a modular decompostion into other components representing her various domestic duties. The wife can observe and issue commands but has little direct involvement with the labors of her station. Within Commander Fred's household, Serena Joy's primary responsibility is to manage the

[14] Not to be confused with Gregory Bateson's use of the term "double bind." While the demands of a project manager could conflict with the standards of the platform, this is not always the case.

Marthas—a class of components that handles the manual labor of domestic upkeep—and engage in sexual intercourse with Fred; however, because Serena Joy is infertile, the labor of pregnancy is decomposed. Offred acts as her womb-module, recognizing herself as encapsulated: the Handmaids are "containers, it's only the insides of our bodies that are important" (96). The modular decomposition of childbearing is most apparent during those scenes in which Fred, Serena Joy, and Offred are engaged in their intercourse ritual, when Offred literally slots into Serena Joy's abdomen. As Serena Joy cradles her, she thinks to herself that this position "is supposed to signify that we are one flesh, one being. . . [Serena Joy] is in control of the process, of the product" (94). Encapsulation is also evident in her relationship with the Marthas, who she claims often "talk about [her] as though [she] can't hear. To them, [she is] a household chore, one among many" rather than a person (48). Even though everyone in the house is effectively isolated from one another, bound by their specific roles in the production of children, everyone is in a sense dependent on Offred's body. The success or failure of Fred's role in shaping Gilead is both symbolically and literally dependent on her ability to bear a child. If she fails, the household will be viewed as defective. Offred may simply be replaced, but given the past failures of Handmaids in Commander Fred's house, he may be declared insufficiently devoted to Gilead's cause and fall victim to a "purging." Modular encapsulation's bound rationality is thus not a simple hierarchy. The cybernetic dimension of OOP permits control to exist at both ends of a hierarchy, leaving those in the middle able to act only within a limited set of circumstances.

III. Conclusion: OOP and Cultural Histories of Software

Atwood's novel illustrates that a "freedom from" approach to complex problems "solves" them by relegating them beyond the scope of a system. OOP, in other words, makes it easier for programmers to read algorithms in source code by filtering their perspective so that they can't, or

don’t have to, focus on aspects of software that are difficult for a human reader to understand. If one’s goal is simply to manage the production of software, then this selective filtering is mostly beneficial. Any control that is ceded to other developers or designers lowers the cognitive burden of everyone involved because encapsulation means that no one needs to know everything about a system for its components to operate together in a well-coordinated fashion. The highest level designers can simply assign tasks to trusted programmers who in turn can draw on platforms of component libraries that represent a demonstrable standard of stability. But what if our relationship to software’s source code isn’t one defined primarily by production? How does our perspective on OOP change if we approach software source code as an aesthetic object subject to cultural study or as a site of political power? Scholars practicing methodologies like Critical Code Studies that encourage us to engage in a hermeneutics of source code and legal activists like Lawrence Lessig who argue that source code is homologous to, and in some cases more powerful than, legal code raise these very questions but they do so without explicitly addressing how OOP constrains our ability to address them.

In many ways, OOP encourages the same sort of myopia that led Henry to refuse to acknowledge that Damon’s death was caused by the very logic he built into the system. By allowing us to read source code that is encapsulated within specific modules, OOP makes this myopia functionally inconsequential. In a software system that is modularized properly, a fixation on a particular algorithm does not matter, even if it leads to a misinterpretation of what was going on elsewhere in the system, so long as that particular algorithm fulfills its assigned role. Models of source code criticism, whether as part of an aesthetic, cultural, or legal inquiry, that privilege close reading assume a model of software design wherein particular algorithms can be directly representative of the entire system. Contrary to Kay’s description of OOP as a form

of recursive self-representation, the cybernetic element of OOP places a significant share of information processing in the structural relationship between components rather than merely in the components themselves. Close reading source code in order to piece together a rich, complex algorithmic narrative of OOP software is possible provided the scale of the source code's text remains human readable. Such a narrative would still functionally resemble the tracings of spaghetti code that procedural programmers dealt with, jumping in and out of various components to follow the flow of information; however, modularization minimizes the effect of the disconnect between human and machine perception on code legibility.[15] Yet modern software is now so vast that these methods fall prey to the same problems raised by engineers during the software crisis. Critics and cultural historians of software, in other words, need to explore methodologies of scale. The remaining chapters of my dissertation serve as an argument that while there are still cognitive barriers to understanding software, it is possible to leverage techniques of source code management and text analysis to frame software as a cultural and political problem.

Nonetheless, problems of access remain. A single software application is connected to hundreds of software libraries. Both the application and the component libraries it draws on are defined independently, and yet during run-time the application is functionally inseparable from the network of relationships it participates in. As in Gilead's households, that application is bound by the algorithms those low-level component libraries provide as support. Companies like

---

[15] See, for example, my study of the SCUMM game engine, "Narrative and Spatial Form in Digital Media: A Platform Study of the SCUMM Engine and Ron Gilbert's *The Secret of Monkey Island*," *Games and Culture* 7.3 (May 2012): 209-237.

Microsoft and Apple thus exert a degree of control over software ecosystems that has been acknowledged in economic histories of computing but that has not been well documented in intertextual studies of software. Chapter 4 examines the Open Source Software ecosystem as an analogy to the closed platforms of Microsoft and Apple to map the cumulative effect of OOP design on software systems and visualize how transparent design conceptually divides a system's interface from those algorithms that actually carry out users' commands and shape their data. Chapter 5 narrows our focus back to a single application, tracing the influence of particular components in order to examine whether the bibliographic history of source code is distinct from changes visible at the interface. In both cases, the software in question consists of millions of lines of code, equivalent in word count to hundreds of novels, and far beyond the idealized critical framework of a single human reader.

Chapter 4 – GNU/Linux: A Tale of Two Transparencies

"Steve Jobs, the pioneer of the computer as a jail made cool, designed to sever fools from their freedom, has died. . . . Nobody deserves to have to die—not Jobs, not Mr. Bill, not even people guilty of bigger evils than theirs. But we all deserve the end of Jobs' malign influence on people's computing. Unfortunately, that influence continues despite his absence. We can only hope his successors, as they attempt to carry on his legacy, will be less effective."

—From Richard M. Stallman's blog, October 6, 2011.

For those hackers and hobbyists who feel scorned by the corporate turn towards transparent design to expand their customer base, the Free/Libre and Open Source Software movement (FLOSS) represents a return to personal computing's roots. FLOSS' calls for a political transparency in computing, designing systems that encourage exploration and modification. The centerpiece of the FLOSS community is the GNU/Linux operating system. Since its initial release in 1991, GNU/Linux has expanded beyond the realm of tinkering and emerged as a powerful, versatile, and stable competitor to commercial platforms.[1] In addition to personal computers, GNU/Linux has been ported to phones[2], video game consoles[3], MP3

[1] Although the operating system is often commonly referred as just "Linux," this chapter will follow the convention of "GNU/Linux" in order to recognize the operating system's origins as part of the GNU Project.

[2] The Android mobile device operating system is built on top of GNU/Linux.

[3] Sony released an official kit for their Playstation 2 game console that allowed owners to install

players[4], DVRs[5], and even alarm clocks[6]. FLOSS' history is foregrounded in the practice of "forking." Because the source code for every component is made available to users, anyone is able to modify the code and create a new branch of development. This practice is permitted under the agreement that users re-release any modifications back into the community as a separate version, acknowledging the original lineage of authors before them. Different versions of the operating system can be found in the form of "distributions," or forks of the entire platform modified and re-packaged according to a group's interpretation of the community's guiding principles[7], a specific set of use practices[8], or simply disagreements over technical

---

Linux if they had purchased the hard-drive add-on. These kits and their included documentation are no longer available; however, the GNU/Linux community put together their own versions. See "HOWTO: How to Install Linux on a Playstation 2 without Sony's Official Linux Kit" at http://kernelloader.sourceforge.net/tutorial/howtoinstalllinux.html

[4] IPodLinux is a distribution for early iPods: http://ipodlinux.sourceforge.net/index.shtml

[5] MythTV is a distribution that turns a personal computer into a DVR: http://www.mythtv.org/

[6] The original Chumby was an Internet capable clock. It's parent company still produces a legacy model: http://www.chumby.com/pages/chumby_one

[7] The Gentoo and Slackware distributions both stress customization; however, Gentoo and Slackware differ on whether automated package management systems impede the users' ability to configure their systems. See: http://www.gentoo.org/ and http://www.slackware.org/

[8] Scientific Linux is a distribution that includes data analysis and visualization software packages as part of its base installation. See: https://www.scientificlinux.org/

aspects of system management[9]. In this sense, the GNU/Linux community embodies much of the chaos of the early market for personal computing; yet unlike personal computing in the 1980s, the GNU/Linux community actively addresses problems of compatibility by encouraging public debate over and review of shared standards. In short, GNU/Linux represents a different response to the same socioeconomic forces that led corporate developers to embrace transparent design's strategies of obfuscation and functional misrepresentation. According to its most outspoken proponents, GNU/Linux and FLOSS embody another model of transparency in software, one resembling the political transparency considered a necessary precursor to democratic societies.

Yet GNU/Linux has received very little attention in studies of digital media, especially compared to video games, electronic literature, or broader theoretical discussions about networks and code. This absence could be attributed to the general public's lack of awareness of GNU/Linux. Prior to Google's incorporation of GNU/Linux into its Android mobile device operating system, there were few ways to purchase a computer system with GNU/Linux pre-installed and fewer still advertisements or other public representations of the operating system. GNU/Linux has enjoyed a quiet acknowledgment among network administrators as the superior choice for servers, supporting the majority of the Internet's websites, data servers, and mail systems. Unlike commercial operating systems, potential users can't stumble across GNU/Linux

---

[9] The Debian and RedHat/Fedora distributions are both known for their stability but Debian is considered the far more conservative of the two, releasing new versions every couple of years. Each established package management systems that have both been widely accepted as standards within the community, serving as the basis for other popular distributions like Ubuntu and SuSE. See http://www.debian.org/ and http://www.fedoraproject.org/

running on display model computers in an electronics store. Nor are there ad campaigns for GNU/Linux. For most people, GNU/Linux is something discovered by word of mouth or first encountered in a laboratory.

During the 1990s, GNU/Linux was regarded as an operating system for programmers by programmers. For much of its early history, it suffered from the same lack of effective documentation that made the first home computers unusable for most people. Even though the Linux Documentation Project began to collect community knowledge as early as 1992 to make the operating system more accessible, GNU/Linux experts were known for their intolerance of users who lacked technical aptitude. It was not unusual on forums for novices to be told curtly "RTFM" in response to questions.[10] The early GNU/Linux community also exhibited a startling degree of sexism, so much so that concerned members created a list of rules as part of the Linux Documentation Project to discourage gender discrimination and sexual harassment.[11] Considering these last reasons especially, it is not surprising that scholars in digital media such as David Golumbia (2010) and Wendy Chun (2011) have dismissed the GNU/Linux model as insufficiently distinct from mainstream computing, noting that in practice it exhibits many of the same critical problems of centralized power and authority. Even GNU/Linux's self-image of resistance is subsumed, Golumbia argues, by the willingness of big tech companies like IBM and Oracle to build their products and services on top of the GNU/Linux's infrastructure, transforming the community into a source of free labor for neoliberal enterprise (147). While it is true that many contributions to GNU/Linux have come from privately funded developers and

[10] "Read The Fucking Manual"

[11] See: http://www.tldp.org/HOWTO/Encourage-Women-Linux-HOWTO/index.html

that its software has been adopted by corporate, university, and government laboratories across the world, the operating system and the software ecosystem it supports remains open through the availability of source code. This single principle continues to draw a diverse set of perspectives into the community and leaves users able to choose distributions that avoid corporate or closed source contributions if they desire. Software licenses like the GNU General Public License ensure that even if corporate developers contribute code to a project, that code must remain freely and openly accessible.

GNU/Linux and FLOSS has already begun to gain attention within the social sciences as scholars there recognize that its openness allows for the study of technological development in ways that the closely guarded secrets of commercial design do not. I contend that GNU/Linux and FLOSS should be examined more closely by academics in the digital humanities as well, not just because their communities and software present an alternative to transparent design but also because the GNU/Linux community's experiments in practices of openness and access can provide examples for the exercise of similar ideals within the digital humanities. For example, Robert Cummings (2006) rightly observes that the coding submission, review, and commit process resembles academic peer review. In addition to his observation, it's worth noting that many of the Copyleft and other Open Access academic publication models were derived from the GPL. Furthermore, Adrian Mackenzie in *Cutting Code: Software and Sociality* (2006) describes GNU/Linux as more than just a technology: it is also a "social arrangement for the ongoing production of code" (81). Participating in the GNU/Linux community by reading and writing code, or even just submitting bug reports and feature requests for those unable to program, produces a "feedback loop between technical performance and performativity. . .[that] triggers constant modulation both in the object itself and the practices associated with that

object" (76). In other words, GNU/Linux and FLOSS encourage a meta-critical engagement with computing power that transparent design actively discourages in its obfuscation of algorithmic processes.

This view of software is precisely the sort of strategy advocated by scholars in Software Studies and Critical Code Studies like Lev Manovich (2001), Matthew Fuller (2003, 2005), and Mark Marino (2006). Yet both of these subfields within digital media studies have thus far been confined largely to experimental genres, video games, and electronic literature. Unlike more widespread yet mundane software like web browsers, window managers, file browsers, or word processors, the genres of software already receiving attention in digital media studies contain an explicitly aesthetic dimension that readily allows for an extension of critical strategies applied to older media. For example, to account for the interactive nature of digital media, the close reading of software often draws upon reflective techniques similar to those found in Reader Response criticism. Scholars pay careful attention to the forms rendered on screen, discussing the possibilities for interaction visible between user and software and interpreting those possibilities through additional critical frameworks.[12] This approach to software is exceptionally good at exploring how users understand their relationship to software, particularly the ways in which the metaphoric representations of algorithms rendered on screen produce engagements among computation, cultural politics, historical moments, and social movements. Nonetheless, it is important to acknowledge that transparency imposes constraints on our ability to engage

[12] For examples of this approach, see Alexander Galloway's *Gaming: Essays on Algorithmic Culture* (2006), Noah Wardrip-Fruin's *Expressive Processing* (2009), or *10PRINT CHR$(205.5+RND(1)): Goto 10* by Nick Montfort, et al (2012).

critically with software.

In her discussion of methodologies in cultural studies of science, Karen Barad proposes that acknowledging material constraints on discourse can be productive. These constraints are important not because they establish some sort of factual basis for the evaluation of discursive statements; rather, this acknowledgement helps us to understand better "the conjoined material-discursive nature of constraints, conditions, and practices" (152). Material constraints such as physical/chemical/biological processes or the availability and capabilities of tools—in this context, the perception imposed on users by transparent design—matter because our intra-action with them continually places us in a particular relationship to our environment. It is therefore important to understand that these constraints are not just a given to avoid "falling into the analytical stalemate that simply calls for recognition of our mediated access to the world and then rests its case" (152). Instead, we must acknowledge that transparent interfaces, as constraints, are themselves part of a technological discourse and are thus also open to observation, debate, and change. In order for us to account for the discursive constraints of transparency, we must set aside the comfortable similarities to older media evident in these explicitly aesthetic genres of software. These similarities must be understood as effects of transparency.

Even as Critical Code Studies calls for this change in perspective, its methods produce a perspective constrained to particular historical moments in the cultural history of computing. Marino argues in his "Critical Code Studies" manifesto that scholars must recognize "that lines of code are not value-neutral and can be analyzed using the theoretical approaches applied to other semiotic systems in addition to particular interpretive methods developed particularly for the discussions of programs." While he describes an aesthetic of source code—complete with its

own formalisms and stylistic play—he also detaches source code from the circumstances of its production and execution. The prevalence of the object-oriented paradigm of programming means that today's software is not only multi-authored, but that meaning lies as much in structural relationships between components as in the lines of code defining individual components. The close reading of source code is therefore a methodology best applied to software with smaller codebases. Modern software applications often have codebases—the entire collection of source code documents describing the program that are required by a compiler to build that program—that rival small print libraries, with word totals equivalent to a handful, dozens, or even hundreds of novels. Beginning the 1970s, computer programmers restructured their practices to account for this scale (see Chapter 3), but digital humanists by and large have not recognized the epistemological problems that scale poses for software studies.

As I discussed in the previous chapter, object-oriented programming emerged in the early 1980s as a solution to the problem of legibility in massive, complex software systems. By writing source code according to the object-oriented of principles of modularity, encapsulation, and polymorphism, programmers design component systems that shape information by passing data between one another. This structural process is similar to the Fuller's description of the way information is generated within media ecologies, where networks of hardware can produce new forms of intelligence that can't be explained by the sum of their parts. A network of technological objects "taken up in unexpectedly massive quantity and variety" affords "further connection to modalities of life and mediality, which it then also becomes folded into and continues to mesh with and compose" (49). In this sense, Critical Code Studies enacts a problem opposite to transparent design and one familiar to post-New Criticism schools of literary theory: too close of a focus on the source code itself removes software from the broader context of

computation. Close reading alone cannot account for the participation of particular lines of code in the abstract representations of information that span multiple components in a software system. Without recognizing object-oriented programming as a bibliographic strategy, scholars cannot fully articulate the relationships that programmers describe in source code between users, their machines, and their data. The close reading of source code is therefore best suited to early forms of personal computing software. Studying the source code to modern software requires a methodology that can account both for its scale and the complex bibliographic relationships between source code files resulting from object-oriented principles.

Fortunately, this problem of archival scale is not unprecedented within the digital humanities. While the aims and techniques of the digital humanities are widespread, recent applications of "distant reading" or large-scale, text analysis have demonstrated that it is possible to identify broad textual patterns across a corpus too large for a single reader to trace. Digital humanists devising ways to adapt information retrieval techniques, like topic modeling, to to identify and interpret discursive elements in large corpora. Topic modeling is an unsupervised machine learning technique that searches for groups of words with a high probability of appearing together in the same document.[13] One advantage that topic modeling has over simpler word or feature counts is that topic modeling can assign different instances of a word to different topics, determining whether that word represents one topic or another based upon the patterns of other words with which it commonly co-occurs. A typical topic modeling output will return a list of words grouped into "topics" and a data vector for each document detailing the number of

[13] The latent Dirichlet allocation algorithm used by most topic modelling frameworks today is explained in more detail in the Chapter 5.

words in each document that belong to each topic. Matthew Jockers has shown that it's possible to trace how literary themes develop over time with topic modeling output by grouping documents by year of publication and graphing changes in topic memberships from year to year.[14] For researchers to use topic modeling in this way effectively, they must possess some prior knowledge of the corpus' context in order to determine whether the topics represent recognizable discursive concepts.[15] Recently, engineers have also proposed that topic modeling can be used to quickly gain a sense of a large code base's organization and have begun to experiment with topic models as a way to trace patterns in software development (Kuhn 2012). Topic modeling, in other words, could derive a brief description of the processes occurring with encapsulated modules that could then be analyzed within a post-processing framework that models the network of relationships between them. Even though the scale and structure of modern source code resist close reading, distant reading allows us to interpret the structures of software that transparency hides from us.

This chapter begins with a historical examination of GNU/Linux and FLOSS as a reaction towards transparency. Richard M. Stallman's description of "free software" argues that the public availability of source code will undo the hierarchal relationship between corporate developers and users. His writings thus push against transparent design's implication that the complexities of computation must remain inaccessible for personal computer software to be of

[14] See also Andrew Goldstone and Ted Underwood's historical study of literary scholarship which applies topic modeling to issues of the *Papers of the Modern Language Association* and going as far back as the 1940s.

[15] See Chapter 8 in Jocker's *Macroanalysis: Digital Methods and Literary History* (2013).

benefit to the general public. In this sense, Stallman presents a different understanding of usability, one defined by open-ended affordances rather than channeled through constraints. In practice, however, Stallman ideas have not been implemented uniformly throughout the GNU/Linux and FLOSS communities. Whereas some GNU/Linux distributions foreground customization and equate access to modification with freedom, others have made great efforts to present themselves in terms similar to commercial operating systems, foregrounding ease of use in ways that bring to mind some of Apple's advertisements. These differences raise concerns regarding whether the source code of GNU/Linux actually reflects the free software's ideals of political transparency and user freedom. Following the historical discussion of GNU/Linux, I conduct a case study of a graphical user interface library called Xfce, as implemented in the Debian distribution of GNU/Linux that examines the structural relationships between package dependencies in a user friendly interface. The results of this study reveal that transparent design in practice is more nuanced and deliberate than the descriptions of as a blanket strategy of simplification as proposed by academic and commercial developers in the 1980s (see Chapters 1 and 2). This case study, in short, shows just how little we can see and describe when we study software solely via its interface.

I. GNU/Linux and Political Transparency

If William Gibson's dystopia is one in which a corporate monopoly on computing power leaves a disenfranchised population no options for access other than to scrounge through last year's techno-junk or steal the hottest hardware, Richard M. Stallman's is one in which corporate technocrats freely share their resources, knowing that all access and use will only occur under their terms. In 1984, the same year that Gibson's *Neuromancer* and Apple's Macintosh were released, Stallman wrote "The GNU Manifesto," publishing it the following year in *Dr. Dobb's*

Journal of Software Tools.[16] Short for "GNU's Not Unix," Stallman described his GNU Project as a reaction to AT&T's decision to restrict access to its Unix operating system through copyright. Although AT&T already exerted some control over access to Unix through license agreements with universities and corporations, it could not profit from the operating system as a result of the settlement terms of a 1956 anti-trust suit; however, in 1983 a second suit resulted in the break-up of the Bell System, separating Bell Labs—the home of Unix within AT&T—from its parent company. Prior to 1983, Unix's licenses included full access to the operating system's source code, making it an extremely valuable tool for universities because it allowed students to study its internals directly and faculty to modify the source code to suit the needs of their computing laboratories. The commercial licenses that followed in the wake of the 1983 settlement removed this access, accelerating the trend of "closed sourcing" among programmers that already had troubled Stallman.[17] But with the GNU Project, Stallman described a future in which users would "no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes" (36). Stallman's writings rarely use the term "transparent," but when they do it is always in reference to political transparency: the idea that the inner workings of a system be made as visible and as accessible as possible to those who are subject to them. Although Stallman's early writings do not address transparent design directly,

---

[16] Unless otherwise noted, all references to Stallman's writings are from his collection of essays, *Free Software, Free Society: The Selected Essays of Richard M. Stallman*, 1st Ed (2002).

[17] For a more detailed account of the Stallman's experiences as a programmer at MIT's AI Laboratory during the beginnings of the commercialization of software, see the final chapter of Steven Levy's *Hackers: Heroes of the Computer Revolution* (1984).

the politically transparent mode of design he calls “free software” can be readily interpreted as diametrically opposed to transparent design’s strategies of obfuscation.

The GNU Project itself did not produce a complete operating system. By 1990, the GNU Project had assembled many of the core system tools required for an operating system, yet it lacked a kernel, a bundle of libraries that carry instructions back and forth between a machine’s hardware and software. In 1991, a Finnish graduate student named Linus Torvalds began a similar projected, frustrated that the Minix operating system, itself a reverse engineered clone of Unix, was available only for educational purposes.[18] Torvalds wanted an operating system he could tinker with outside of his university’s laboratories, and because the GNU Project had not completed its Unix clone, he began to write his own. By the end of the year he had released his “Linux” kernel as free software. In 1992, his kernel drew the attention of developers working on Stallman’s GNU Project, who began to integrate the Linux kernel into the system tools they already had completed.

Today, the GNU/Linux operating system is still very much a combination of the two projects, at least at its lowest levels. Stallman’s Free Software Foundation still oversees development of the GNU Project’s tools, and Torvalds still maintains the code base for his kernel. The prevalence of forking and the ability to patch—or insert modifications as the source code is built into executable files—means that Stallman’s and Torvalds’ projects maintain canonical versions of their software which may not match their implementation in the various GNU/Linux distributions in circulation. Yet this history very much resembles the free software

[18] This account is taken from Torvalds’ memoir, *Just For Fun: The Story of An Accidental Revolutionary* (2001).

that Stallman describes in his writings. The ready availability of source code allows others in the community to modify both the GNU tools and the Linux kernel to suit the needs of their own projects. Stallman and Torvalds do not always agree with one another about the nature of free software, nor with other prominent members of the community. Keeping the source code freely available, however, allows for a mode of design defined by political transparency: because the core components of GNU/Linux are free software, the community ultimately has the final say in how they are used. Yet the community is more or less a loose confederation of the like-minded, with each project fork determining its own system of governance and decision making.[19]

Stallman's work shares with the radical technologists of the late 1970s and early 1980s the belief that personal liberty and empowerment is possible through technology.[20] Yet unlike many of his contemporaries, Stallman did not go on to found a technology company, and in fact his work is characterized by a distrust of privatized development. In his "GNU Manifesto," he argues that allowing for the free circulation of source code "will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you" (37). Monopolistic control over software, he

---

[19] Many projects include an explicit statement of rights and responsibilities. See, for example, the Debian Constitution: https://www.debian.org/devel/constitution

[20] For a more in depth look at the revolutionary tone surrounding personal computing, see Fred Turner's *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network and the Rise of Digital Utopianism* (2006).

notes, establishes a technological "police state" in which corporate developers "force everyone to obey them," building software that permits only their models of use (36). When users are not free to determine their own relationship to computers, software becomes purely a way of "[e]xtracting money from users of a program" (38). Establishing a single, correct mode of use is "a deliberate choice to restrict" agency that results in "the deliberate destruction" of the possibilities of creative participation in technology, thereby "reduc[ing] the amount of wealth that humanity derives from the program" (38). Because operating systems establish a common platform for software across a variety of hardware combinations, they should not be designed in ways that constrains the possibilities for techno-cultural production. In other words, allowing a single company to control their development, in turn, allows a small group of people to shape all software written for it.

In this regard, Stallman was one of the first to propose that software serves an institutional function, constraining computer use within an ideological framework of laws and policies described within its source code. Legal scholar Lawrence Lessig cites Stallman's work as an influence on his own activism. Although Lessig is known most for his argument that "code is law," he comments in the introduction to Stallman's collected writings that he first encountered the idea in Stallman's work. In *Code and Other Laws of Cyberspace* (2006), Lessig explains that like laws, software "determines what people can and cannot do" with their computers; source code, he continues, "codifies values, and yet, oddly, most people speak as if code were a question of engineering. Or as if code is best left to the market" (77-78). Yet unlike the law, software does not prescribe what is allowed or disallowed through threat of punishment; instead, software's influence is more or less Althusserian in that the rituals of use it provides users inculcate in them attitudes about technology. Through continued interaction, users learn not

only how a system "should" be organized, developing the intuitive understanding that transparency's theorists claim software provides *a priori*, but also the "correct" attitudes they should have about their machines and those who design them. Software is therefore more powerful than law in that its influence is constant during the act of use, and transparent design encourages users to ignore that influence and to internalize it blindly as a condition of access. Although Stallman frames his GNU Project in response to the closure of Unix, it's important to recognize that his announcement occurred during a moment in the cultural history of computing when Steve Jobs, as part of the campaign to promote Apple's Macintosh, was commenting in interviews that his company knew more than anyone else what users wanted out their computers, that Apple's view was "the right way of looking at products" (243). Given that Unix was not an operating system intended for home use, it's not surprising that Stallman's manifesto does not mention Apple, Microsoft, or IBM explicitly; yet he is clearly opposed to the relationship between programmer and user that results from transparent design. The dystopian imagination underlying his description of a politically transparent mode of software challenges Apple's self-promotion of the Macintosh—twisting the smiling face on its startup screen into the face of Big Brother that its avatar smashes in the 1984 Super Bowl commercial—and Microsoft's quiet monopolization of the operating system market through the widespread adoption of DOS.

For software to be "free," Stallman argues, it would have to be recognized as a cultural discourse rather than simply as a technology; however, this stance is difficult to realize if only a small number of people are allowed to participate in that discourse. Stallman's 1996 essay, "Free Software Definition" clarifies many of the ideas he first proposed in "The GNU Manifesto." He begins by noting that free software should be understood as "a matter of the users' freedom to run, copy, distribute, study, change, and improve the software;" in this respect, it's best

understood to be "'free' as in 'free speech,' not as in 'free beer.'" (43). The free distribution of software, he continues, is possible only as a consequence of the freedom to study and modify source code. In outlining the four basic properties of free software, he also begins to shift away from his earlier emphasis on personal liberty and more towards software as a common platform for a community:

- Freedom0: The freedom to run the program, for any purpose.
- Freedom1: The freedom to study how the program works, and adapt it to your needs. (Access to the source code is a precondition for this.)
- Freedom2: The freedom to redistribute copies so you can help your neighbor.
- Freedom3: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. (Access to the source code is a precondition for this.) (43)

Furthermore, Stallman argues that the freedom to participate in software as a discourse should take precedence over any technical concerns. In "The GNU Project" (1999), an essay reflecting on GNU's role in shaping the Linux operating system, Stallman addresses the concern that GNU/Linux will never force companies that follow the Apple model out of the market. He argues that even if free software does not have any technical advantages over commercial software, it would still have "a social advantage, allowing users to cooperate, and an ethical advantage, respecting the user's freedom" (24). While the writings on the GNU Project may not effect the same sense of urgency as novels by science authors like Gibson, Bruce Sterling, and Margaret Atwood—all of whom envision a future where all political and social power is based upon the control of information flows—Stallman similarly presages many of the technologies present in modern software used to extract money from users or monitor their behavior. Today,

app stores and digital rights management tools continually examine users' computers, ensuring that everything has been paid for properly and is accessed only by the original purchaser. Upgrades, add-ons, and enhancement packages are sold in the form of "unlocks" that enable already features already present on users' machines but locked until a payment is received to encourage further transactions with developers beyond the initial purchase. Cloud services offer users access to their data from anywhere with an Internet connection, usually at the cost of analyzing anything they upload. In a future where transparent design runs rampant, users adopt these new technologies swiftly with little consideration of the socio-economic and political dimensions of the algorithmic operations obfuscated by user-friendly interfaces.

In order to understand the significance of Stallman's claim that software can be politically free only if its source code is made freely available, it is important to be familiar with the functional relationship between software and source code. Today, commercial software is distributed as executable binaries without any reference to the original source code. This process is referred to as compilation, and it involves organizing all documents associated with the software in a way that reflects the intertextual relationships produced via information flows between separately defined components. Once organized, these documents then systematically translate them into a low level language of direct hardware instructions, usually some form of Assembly. The Assembly code is then translated further into a binary code executable on a specific type of hardware platform. In short, compilation produces a set of files that can be executed independently of the original source code and from which the original source code cannot be recreated. Although commercial developers obtain copyrights to protect access to source code, the technical constraints to access posed by compilation make them largely a formality. In fact, it is much more common for developers to use broadly written patents on

algorithms to prevent competitors from producing similar software. When software is distributed without its source code, the possibilities for modification and customization are few. Some software is built with an application programming interface (API) that allows users to design their own modular extensions or plug-ins to add extra functionality to a program; however, APIs often leave core features of the software unchanged. Apart from APIs, the only possibilities for modification exist in the form of security exploits that hackers use to inject their own code into the already running software, a task that only an exceptionally skilled minority of programmers have the ability to undertake.

Without access to source code, even the most skilled programmers are censored by compilation. In effect, free access to source code would make it possible to upend the hierarchal relationship of programmer and user. The free software community that Stallman hoped to build around his GNU Project is based on his experiences working in the Massachusetts Institute for Technology's Artificial Intelligence Lab. Stallman recounts in a speech, "Free Software: Freedom and Cooperation" (2001), that the lab's computers used an operating system developed in-house in the 1960s dubbed the "Incompatible Timesharing System." While the operating system itself was technically influential for its time, the lab's policy of making all parts of the software available for review and modification by the lab's researchers was what impressed Stallman the most. Although this freedom likely arose as a result of the lab's open-ended experimental culture, Stallman notes it also encouraged researchers to improve the lab's computer systems whenever they encountered some aspect of the lab's software that interfered with their work. His first awareness that compilation could serve as a form of censorship came when the lab obtained one of the first laser printers from Xerox. Wanting to add a simple alert to report when the printer had jammed, he found that the drivers had been included only in pre-

compiled binary: “Xerox wouldn’t let us have the source code. So, despite our skill as programmers–after all, we had written our own timesharing system–we were completely helpless to add this feature to the printer software” (160). In terms of the freedom of speech, any value a speaker might add to a discourse is moot if he or she is not permitted to speak. For Stallman and others at the lab, sharing and altering software functioned as a form of discourse, a politically transparent exchange of ideas involving the citation of and disagreement with the work of others with the end goal of producing something of valuable for the community.

Today, the terms “open source” or “open access” are used much more frequently than “free software” or “free access” to describe the community that Stallman wanted to build. The now complicated FLOSS construction used to refer to the community is an attempt to account for its various views—“free” to acknowledge its roots in Stallman’s work, “libre” to emphasize a particular interpretation of free (as opposed to “gratis”), and “open source” to acknowledge efforts to recast the community as more apolitical and business friendly. Although there was considerable interest in Stallman’s GNU Project and its operating system, many were concerned that his politics would prevent widespread adoption of GNU/Linux and applications written for it. In 1998, Bruce Perens and Eric S. Raymond founded the Open Source Initative (OSI) in response to the Netscape Corporation’s decision to make the source code for its popular web browser, Navigator, publically available in order to

> harness the creative power of thousands of programmers on the Internet by incorporating their best enhancements into future versions of Netscape’s software. This strategy is designed to accelerate development and free distribution by Netscape of future high-quality versions of Netscape Communicator to business customers and individuals, further seeding the market for Netscape’s enterprise

solutions and Netcenter business.[21]

The OSI's website states quite explicitly that the organization was founded to temper the politics of Stallman's FSF, noting that the founders "believed that it would be useful to have a single label that identified this approach and distinguished it from the philosophically—and politically—focused label 'free software.'"[22] The "pragmatic, business-case grounds" that the OSI cites in Netscape's decision are not specifically stated in the company's press release; however, given that their decision was just months before the United States filed its anti-trust suit against Microsoft in May 1998, it is apparent that Netscape open-sourced its browser suite as part of a last ditch attempt to compete with Microsoft's monopolistic bundling of Internet Explorer and other competing web-technologies into its Windows operating system (see Chapter 5). Netscape's decision and OSI's goals initially seems to realize Golumbia's fear that FLOSS is simply a source of cheap, or even gratis, labor; however, Netscape failed to commercialize the work of the community because there was little incentive for users to pay for the "high-quality versions" of their browser when the open sourced "Mozilla" fork worked just fine. Development of Firefox, the direct code descendant of Netscape, is now overseen by a non-profit foundation that is funded through donations and advertisement royalties received via the search engines

---

[21] The text of this press release taken from the blog of Mitchell Baker, "Chief Lizard Wrangler" (Chairperson) of the Mozilla Foundation. The original announcement was taken down after American Online purchased and later closed Netscape's offices. She authored the legal framework for the open sourcing of Netscape as Mozilla. https://blog.lizardwrangler.com/2008/01/22/january-22-1998-the-beginning-of-mozilla/

[22] This and other remarks regarding the OSI's history taken from http://opensource.org/history

integrated into the browser's interface.

To protect free software, and further distinguish it from similar movements that called for the release of source code, Stallman authored the GNU General Public License (GPL). A model for later copyleft licenses, the GPL functions by first declaring copyright over a piece of software and then selectively abdicating certain authorial protections, allowing for modification and redistribution. Under the terms of the GPL, all software using free software components and all forks of free software codebases would themselves have to be released under the GPL. Additionally, all components supporting a piece of free software would themselves also have to be protected under the GPL. These last points are a source of disagreement within the community, a place where the GPL and the OSI's "open" license diverge. In practice, OSI-approved licenses have not prevented open source software from being incorporated into obfuscated software. Beginning with MacOS X in 2000, Apple's operating systems uses some of the software comprising the FreeBSD operating system via the NeXTSTEP fork that Jobs worked to commercialize after his dismissal from Apple in 1985. To their credit, Apple has released the Unix-like portions of OSX as "Darwin" under an open source license; however, the majority of OSX remains closed, including middle layers that automate and control the underlying Darwin in service of Apple's transparent interface. The availability of source code in this instance means little as the majority of OSX is obfuscated. More recently, Google's Android mobile operating system incorporates elements of the GNU/Linux operating system. Android's source code has also been publically released, but community built forks remain separate from the commercial code-base. Users wanting to try one of the community forks must "jailbreak" their devices, voiding their warranties in the process. Given that the transparent design of mobile hardware makes these devices both fragile and difficult to repair, community forks are a risky

proposition for many users. Regardless of the political sentiment behind a piece of software, it can still be obfuscated by release models that privilege small groups of programmers. In practice then, the availability of source code alone does not necessarily guarantee a freedom from the proprietary models of control that support transparent design.

The ideological conflict between free and open source reflects a larger debate across the GNU/Linux community about a potential incompatibility of political and technical goals. The OSI's move to de-politicize free software represents a belief that because most users are accustomed to a certain degree of obfuscation through regular exposure to transparent technologies, the significance of Stallman's political transparency will not reach them. For his part, Stallman acknowledges that the most open source licenses were written with the same ideas as his own, many in effect tweaks of his GPL. In an essay written following the foundation of OSI, "Why 'Free Software' is Better than 'Open Source'" (1998), he comments that the FSF and OSI "disagree on the basic principles, but agree more or less on the practical recommendations. So we can and do work together on many specific projects" (57). The problem, he continues, is that OSI's licenses permit authors to continue to exert control over their source code after releasing it rather than treating it as public good. Whereas Stallman wants users to be able to assume the role of programmers, the OSI's more commercial rhetoric quietly reinforces a technocratic power by placing more emphasis on open source as a way of giving technology professionals a new strategy to gain the trust of consumers.

While it might be hard to imagine how this conflict looks within the context of a single application, the availability of GNU/Linux's source code provides us with a unique opportunity to see how the political relationships take shape in practice. Torvalds states in his memoir that although he shares with Stallman the view that the commercial control of software is a form of

"despotism" that can only be avoided if software is open sourced and freely modifiable, he states that Stallman too often sees "everything in black and white" and is unwilling to compromise (215-218; 195). In contrast to Stallman, Torvalds' idea of freedom is more general, not something produced through legal protections, and the advantage lies in increasing the possibilities for innovation rather than in transforming software's politics. For Torvalds, the benefits of freely accessible source code are primarily managerial: "By not controlling the technology, you are not limiting its uses. You make it available and people make local decisions—to use it as a launching pad for their own products and services. And while most of those decisions don't make sense in the larger scale of things, they actually work really well" on a local level (229). Because the GNU/Linux operating system is designed using object-oriented principles, each component within it represents a programming team's distinct interpretations of free and open source software. The relationships they form with other teams is therefore visible in a network of informational relationships between the operating system's components.[23]

II. Examining GNU/Linux: A Case Study

The GNU/Linux operating system is a complex eco-system, with popular distributions exerting influence locally over their users and platform libraries quietly shaping the projects built from them through the intertextual networks of dependencies that branch off from them. According to Distrowatch.com, a website dedicated to tracking the development of GNU/Linux distributions, there are currently 291 active distributions.[24] This number doesn't include those

[23] See for example: http://techcrunch.com/2012/04/19/an-interview-with-millenium-technology-prize-finalist-linus-torvalds/

[24] Statistic retrieved 8 Mar 2014.

that ceased development, those that aren't yet publically available, nor the various "spins" released by major distributions that include different default system configurations. Each of these distributions share a common ancestry that can be traced through a history of forks, and all of them incorporate the base GNU compiler libraries, GNU userland tools, and Linux kernel.[25] In practice, however, not all of them acknowledge free software: some only in notifying users of the terms of the GPL when required; others clearly labeling which portions of the system are free software, which use alternative open source licenses, and which incorporate proprietary libraries; several install only free software by default but explicitly present users with the option to install non-free packages; and a few install or make available no non-GPL components by default. Even though every distribution of GNU/Linux benefits from the labor of Stallman and his GNU Project, more recent distributions also reflect the influence of commercial design's obfuscation in the interest of usability.

Stallman's FSF maintains a list guidelines for groups looking to create distributions that only use free software, as well as a review of what the most popular distributions should change in order to comply with these guidelines.[26] The FSF's review represents one effort to evaluate the political transparency of distributions by examining the mixture of free and non-free software; however, there has been no such effort to evaluate the prevalence of transparent design practices in GNU/Linux. Although the claim that this year will be "the year of the Linux desktop" when GNU/Linux finally surpasses Windows and OSX's marketshare is a common

---

[25] For a high resolution diagram tracing the ancestries of distributions: http://futurist.se/gldt/

[26] For the guidelines, see: http://www.gnu.org/distros/free-system-distribution-guidelines.html For the FSF's review of distributions, see: http://www.gnu.org/distros/common-distros.html

joke among the tech journalists, there has nonetheless been considerable effort by companies like Canonical, who privately funds the development of the Ubuntu distribution, to make the operating system more widely appealing and user-friendly. For many outside of community, Ubuntu has become the face of GNU/Linux through its embrace of the familiar rhetoric and programming strategies of transparent design.

It's important to note though that in addition to making personal computers "easy to use," transparency also permits practices like datamining and copyright monitoring to occur unseen. Unless developers choose to share their source code or even just their pre-design white papers, they are usually only discovered by hackers.[27] There is no obligation for hackers to reveal their discoveries to others, and they often keep discoveries to themselves in order to exploit less technically aware users.[28] Recently, Canonical's development team embedded datamining

---

[27] For example, see the anonymous "Firewall Your iPhone" in the July 2012 issue of *2600*, in which a hacker discovers that his iPhone is continually sending small pieces information to Apple's servers even when he had his Internet connection disabled via the options available to him through the iOS interface.

[28] Kyle McDonald's digital media project "People Staring At Computers" is an example of one of the more benign exploitations of Apple's transparency. While McDonald was required to remove the project from public circulation, *Wired*'s coverage of the events explains the project in detail. Luis Mijangos similarly exploited transparency to prey upon women through their laptop cameras. See: http://www.wired.com/threatlevel/2012/07/people-staring-at-computers/all/ and http://arstechnica.com/tech-policy/2011/09/how-an-omniscient-internet-sextortionist-ruined-lives/

software into their Ubuntu distribution. Rather than manually searching through lists of programs or directories of files, many users now locate programs and data through local search utilities built into an operating system's GUI. Beginning with Ubuntu 12.10, commands issued to the local search utility were also routed to Canonical's servers and then through to Amazon's product database.[29] In addition to receiving results from their hard-drives, users were also presented with some of the results from an Amazon query, forwarded to them by Canonical. The incident drew a variety of reactions, from hardline free software supporters saying that the spyware should be removed entirely to more tempered open source voices arguing that the spyware would be fine if users were asked explicitly while installing Ubuntu if they were willing to have their searches shared. Others defend Canonical's practice, noting that routing the searches through Canonical's servers effectively made them anonymous, as the searches could not then be linked by Amazon to particular users. The feature, they conclude, is necessary to help further fund the operating system's development since Amazon likely pays generously for the anonymized search data. In a broader context, GNU/Linux's first incident of spyware suggests that even though the operating system emerges out of a movement opposed to transparent design, there is nothing inherent in its design that prevents its community of developers from employing strategies of obfuscation or functional misrepresentation.

Within the context of this study, the incident suggests that GNU/Linux is a valuable case study for tracing the way that modern, user friendly operating systems are structured to support transparent design. While it is possible that Canonical incorporated transparent design practices into Ubuntu independently of the rest of the GNU/Linux ecosystem, Ubuntu does share a large

[29] See http://www.omgubuntu.co.uk/2012/10/does-ubuntus-amazon-lens-break-eu-law

portion of it's codebase with its most immediate ancestor, the Debian distribution: a popular choice among university laboratories due to its stability and meticulous documentation. The ability to access the source code for GNU/Linux affords an opportunity to examine the structure of operating systems, something not possible with commercial systems. If the "freeness" of software were measured on a continuum, Debian would rest in the very middle. By default, Debian installs only free software but prompts users during installation if they want to allow closed libraries. Because Debian can be configured readily into either a strict free or a more permissive, non-free/commercial form, it can allow us to see how an operating system's components can be exploited in the service of transparent design's obfuscation and test to what extent free software's politically transparent ideals are reflected within the software itself.

Previous chapters have thus far discussed transparent design in abstract, theoretical, and historic terms. When considering how its principles take shape in an example of operating system software, transparent design should be understood as a structural relationship between higher-level applications and the lower-level libraries that support them. Figures 4.1 and 4.2 illustrate the layered model of system design and how transparency affects users' perception of those layers. Within digital media theory, N. K. Hayles' "flickering signifier" is perhaps the most well-known representation of the layered model. In *How We Became Posthuman* (1999), Hayles describes a "flexible chain of signifiers. . .[i]ntervening between what I see and what the computer reads" composed of "the machine code that correlates the alphanumeric symbols with binary digits, the compiler language that correlates these systems without high-level instructions determining how these symbols are to be manipulated, the processing program that mediated between these instructions and the commands I give the computer, and so forth" (31). Hayles' description generally matches those found in engineering manuals. For example, in Andrew

Tanenbaum explains in *Structured Computer Organization* (5th Ed, 2006) that layer models function by conveying instructions to and from a computer's hardware. For this process to work, the layers "must not be 'too' different" as each translates or interprets those adjacent to it; although generally speaking, lower layers are more direct descriptions of hardware operations, and the most abstract higher layers must roughly correlate to the functions of lower layers for the system to operate correctly (2-3). While Tanenbaum is correct in noting that functionally the layers must work in sync for users to operate it, Hayles' description hints at the way transparency affects layered systems. She notes that the "longer the chain of codes, the more radical the transformations that can be effected" (31). In other words, only the most adjacent layers must correlate with one another, even then only minimally. Transparency thus represents a radical transformation between layers, where the interface reinterpreting the users' commands in terms that will be understood and defined differently by the layers below it.

Transparent design's obfuscation complicates functional accounts of layered software architecture because they often assume a series of faithful translations. In *The Language of New Media* (2001), for example, Lev Manovich uses the term "transcoding" to describe the rendering of signifiers, whether they are textual, visual, auditory, or some combination of all three. According to Manovich, all digital objects have two sets of signifiers: a "cultural layer" which represents hypermediated forms of older media and a corresponding "computer layer" which manages cultural representations using its own set of machine signifiers on which it enacts a transformational logic to produce the cultural layer (45-48). Like Hayles, Manovich invokes the layered model to argue that new cultural forms are made possible by representation of older media in the computational layer. Yet Manovich implies that there is a direct correspondence between the forms visible on screen and the data forms managed by computation that users can

recognize in order to take advantage of algorithmic processes enveloping new media forms—in his case, digital images and video—simply by comparing them to the limitations of older forms. As Figure 4.2 shows, this correspondence is not guaranteed when using a transparent interface. Manovich's model is valuable in drawing our attention to the way humans and computers understand something like a digital image different, the former as a visual artefact and the latter as a grid of pixel definitions, but it collapses the complex network of intertextual dependencies underlying the interface into a single, simplified layer. Transparency in this sense simultaneously a representational disruption and a functional continuity between the cultural layer and the complex networks of the computation beneath it. Interrogating layered architectures therefore requires tools that can derive the conceptual content of software—both the functions performed and the language used to represent them—and that can use those concepts to map the structural relationships among and between an operating system's layers.

III. Description of Dataset and Methods

GNU/Linux is valuable as a case study for operating system structures because many distributions, including Debian, provide tools or databases that meticulously document software dependencies for every component that users can install. Building a dependency tree of a GUI software will thus trace the network of components that must be in place during runtime for users to interact with the interface. For this study, I chose Xfce, a light-weight GUI based on the GTK+ graphic toolkit available through Debian's repositories in either binary or source code form. Unlike the two most popular GUIs in Debian, KDE and GNOME, the entirely of Xfce is available as source code and largely comprised of a single programming language.[30] Examining

[30] KDE includes some optional non-free libraries and GNOME is written in at least four different

the relationship between the layers of Xfce's dependency tree should therefore allow us not only to expose the complex structures of modern application software that are not accessible on commercial systems but also reveal in the transformations between its lower libraries and its more abstract, interface layer the structural features that transparent design exploits.

My workflow required two data-gathering stages to map the Xfce dependency tree and then acquire and organize the source code to match its structure. Following the initial data gathering stages, I planned and implemented two analytical stages to topic model the source code model and then analyze that model against the dependency tree. In the first data gathering stage, I wrote Python scripts to mine Debian's software repository index for components definitions—called "packages" in Debian—and then to trace dependencies beginning with the package designated in documentation as Xfce's installation package: "xfce4." Debian's definitions make a distinction between required and recommended dependencies (or optional features), but the scripts used for this project did not acknowledge that distinction. My scripts then wrote the dependency tree in the Pajek network format with dependencies marked as arcs (directed links). Next, I used Pajek's acyclic hierarchy algorithm to eliminate loops within the tree in the event of packages which were mistakenly listed as dependent upon each other. Finally, I used Pajek's depth calculation algorithm to partition the network into layers. In order to calculate "depth," Pajek traverses a directed, acyclic hierarchy in order to determine the maximum number of links between the top level node and all other members of the network. The resulting layered network is visible as Figure 4.3.

---

programming language. Xfce is a mixture of C and C++, which are similar enough in syntax to be analyzed using the same toolset.

For the second data gathering stage, I used Python scripts to read the tree produced during the previous stage and download the source code for each component. These Python scripts make use of Debian's command line software repository interface, "aptitude," to automatically download and extract each collection to a temporary directory and apply Debian's official patches. Once acquired, the Python scripts located all files present in the extracted directory that contained C or C++ code and transferred them to working directory for pre-processing. During pre-processing, all C and C++ code files were stripped of non-alphabetical characters. Following the recommendations of Adrian Kuhn, et al. (2006), my scripts then tokenized all commands and variable names by splitting them according to conventional naming practices, specifically the use of underscores and camel-case to form compound terms. Language primitives, operators and standard commands were treated as stopwords at this phase of pre-processing and filtered as the separate source code files for each package were merged into a single document. To account for context specific stopwords in Xfce's codebase, I again followed the recommendations of Kuhn, et al. and used a Python script to stem all tokens, count the number of documents each token appeared in, and filtered all documents for tokens that appeared above a certain threshold. After several analytical trials, I set the threshold at 60%. This threshold was also able to remove the GPL, LGPL, and other licenses disclaimers that do not describe the conceptual content of files but are present at the beginning of most GNU/Linux source code files. This threshold also eliminated commands common to most GNU/Linux applications but which are not part of the C and C++ languages themselves. The resulting dataset represents the conceptual content of each package in Xfce's Debian dependency tree. Table 4.1 shows information about the dataset produced by the second data gathering stage.

For the first analytical stage, I wrote a Java interface for the Dragon Natural Language

toolkit that topic models the Xfce dataset. My topic models were generated using 1,000 iterations of Dragon's Gibbs latent Dirichlet allocation (LDA) algorithm. My initial trials used 500 topics, following the example of studies by Tse-Hun Chen, et al. (2012) and Stacey K. Lukins, et al. (2010). While I determined that running LDA with a higher number of topics than documents works well at producing topics that accurately describe the conceptual content of each software package, it proved ineffective in showing how packages share concepts. Coarser analytic trials that generated fewer topics than total documents began to show clustering where packages known in advance to be fulfill similar roles had large percentages of their tokens placed within the same topics. Trials producing 50 and 100 topics worked well in this respect, but trials under 50 did not make sufficient distinction between those packages known in advance to fulfill different roles. The results discussed below were produced from models of 50 topics.

Finally, for the second analytical stage, I used Python scripts to measure relationships between packages by comparing the vectorspace topic models produced by my Java implementation of the Dragon Natural Language toolkit produced during the previous stage. In these vectorspace models, each document is assigned a row and each column in that row represents the number of tokens in that document assigned to each topic. Vectorspace document comparisons based on token frequency commonly use cosine similarity or the Pearson correlation coefficient, both symmetrical functions. These two functions measure various patterns of relationships between packages throughout the dependency tree; however, I found that the results from Pearson are slightly more conservative.[31] To examine similarity between

[31] Globally, both metrics had very similar results. The standard deviation for cosine similarity when the topic vectors of all packages were compared to all others in the tree was 0.1680. For

layers, I used Python scripts to sum the topic vectors for each package located at a given depth on the tree produced during the first stage of data gathering. I then calculated similarities between each layer and between the interface layer and all other layers. Additionally, I calculated similarities between all packages in the dependency tree in order to produce a second network diagram with edges (undirected links) representing each Pearson result above 0.3. This network diagram preserved the layers of the dependency tree in order to visualize potential conceptual relationships between packages not represented in Debian's package definitions. Although I chose this threshold somewhat arbitrarily, 0.3 is generally considered to be the lowest end of a "moderate" correlation between two documents. I also used this set of network-wide similarity measures to construct a new network based on the strongest similarities between components. Finally, in order to examine the presence of specific roles within all three networks, I manually reviewed the key terms assigned to each of the 50 topics and coded each to one of six categories. These six categories include: user applications, GUI and graphics libraries, multimedia encoding and decoding, data-processing applications and libraries, hardware and file system management, and command line utilities. Examples of topic codings can be found on Table 4.2. After coding each topic had been assigned a code, I then used a Python script to incorporate the codes as node colors into the graphs I previously generated.

IV. Results and Observations

My results first show that topic modeling is capable of accurately identifying correlations between packages known to be strongly related. Table 4.3 shows a sample of Pearson results

---

Pearson in the same set of comparisons, the standard deviation was 0.1621. For comparisons only to a package's immediate dependencies, the standard deviations were 0.3652 and 0.3586.

between example packages from different roles in the xfce4 ecosystem and their direct dependencies according to Debian's package definitions. Generally, Pearson correlations between 0.5 and 1.0 or -0.5 and -1.0 are considered "strong." Although not all dependencies showed strong correlations, dependencies known to be strongly incorporated into the construction of higher level packages produced results as high as 0.999. For example, all of the packages responsible for rendering the Xfce environment shared strong correlations with libgtk2.0-0, the GTK+ graphics library, a toolkit with instructions for drawing interface objects like windows, toolbars, dialog boxes, etc. Dependencies that perform similar functions also showed strong correlations. For example, the Hardware Abstraction Library (hal) package is responsible for detecting and maintaining a list of active hardware; the packages responsible for universal serial bus (libusb) performs a similar function by assisting in communications between USB devices the kernel stack. The libusb package is listed as one of hal's dependencies and produced a correlation of 0.997. Dependencies produced by the same project team, such as those written by the Xfce team or the GTK+ team, also showed strong correlations. If topic modeling can highlight known or expected relationships defined by the dependency tree, it's also possible that it can reveal other patterns between packages and layers.

Mapping the coded topic partitions onto those packages that contained source code, Figure 4.4, shows that the various roles identified by topics are fairly evenly distributed throughout the tree.[32] This even distribution of package types visible in the rest of Figure 4.4

[32] The original layers of the Figure 3 have been preserved, but horizontal positions have been optimized to show clustering. Because Layer 1 contains no source code, Figure 4's topmost layer corresponds to Layer 2 in Figure 3.

could be an effect of the coarseness of the topics produced when only 50 are generated. Noticeably absent in Figure 4.4 when compared to Figure 4.3, however, is the top-level package recommended in Debian's documentation for Xfce installation that served as the point of origin for dependency tree generation. According to Debian's documentation, "xfce4" is a "virtual" package because its definition includes no files, just links to the other Xfce packages that appear on Layer 2. Furthermore, packages on Layer 2 are all coded as either "User Applications" or "GUI and Graphics Libraries," meaning that their top topic keywords all described abstract user interface objects—like buttons, windows, or menus—or more primitive drawing objects—like shapes, lines, colors, or shaders. Users' experience of the GUI is in this sense also virtual, as the Xfce GUI does not exist as a singular piece of software but rather as a collection of software objects that work together to draw a unified environment on screen. The mixture of system management packages with graphics libraries and GUI toolsets suggests that there is no clean separation between the computational and cultural layers. Nonetheless, the only layer free of system management or data encoding packages is the highest one containing source code, Layer 2. Even though graphics libraries and other tools used to render the interface can extend several layers deep, the packages that are coded as representing lower level functions like System Management or Data Processing—with topics describing hardware, file systems, or data types—are not directly accessible from Xfce's virtual interface. Access to them is always mediated by the rendering packages on Layer 2. In short, Figure 4.4 represents a structural organization that makes transparency's strategies of obfuscation and functional misrepresentation possible.

Analyzing the relationships between layers shows that the transformations between layers

are not uniform. Figure 4.5 shows Pearson results for comparisons between adjacent layers.[33] Summing the topic vectors to compare layers reveals a sharp drop in correlation below Layer 5, the lower layer to contain a package strongly represented by a topic representing abstract user-interface concepts. Although it is possible that this trend reflects the changing size of each layer, the largest difference in layer size—between Layers 3 and 4—occurs at one of the smaller drops in correlation. Within a layered system, some difference between layers would be expected as packages fulfill different roles or distinct aspects of the same role; however, according to Tanenbaum's definition of layered architecture, some degree of similarity must exist between layers for data to be passed between and transformed by them. The sharp drop between Layers 5 and 6 is thus suggestive of transparency's potential to radically transform data, passing it along in a form that remains function but re-representing it conceptually. Figure 4.6 repeats the layer comparisons by comparing all layers to Layer 2, the packages that comprise Xfce's virtual interface. Pearson results from comparing each layer to the interface again show a significant drop, again between Layers 4 and 6. The sharp increase in correlation at Layer 7 is due to the presence of libgtk2.0-0, a graphics library that most packages in Layers 2 and 3 are dependent on; however, even if libgtk2.0-0 is removed as an outlier, the Pearson results still shows a rise in correlation between the interface and Layer 7. Figures 4.5 and 4.6 thus suggest that the neat separation Manovich and even Tanenbaum see between layers of software does not exist. Rather, software's architecture is more complicated; software's network of dependencies entangle its

[33] Comparisons began to produce outliers below Layer 12. These irregularities are likely the result of the small number of packages below Layer 12. All comparisons discussed below are within the range of Layers 2 – 12.

layers. Transparency, as a consequence, may not simply be the filtering of everything below the interface from the user's awareness. Instead, programmers can apply its strategies of obfuscation and functional misrepresentation to target only specific pieces of software's internal structure.

To further test the possibility that transparent design could work as a targeted strategy rather than a blanket principle, I made global comparisons between packages throughout the entire corpus. One limitation of examining an operating system through a dependency tree is that it is difficult to identify concurrent package use. Xfce's virtual interface, for example, is the product of the packages on Layer 2 working in tandem. Even though the dependency tree does not define any direct relationship between those packages, they nonetheless function together and users would not experience the familiarity of a desktop GUI if they loaded only one or two of them. In turn, many of these packages are drawing on the same set of dependencies to carry out commands issued to them by users. These packages are used concurrently but are related only indirectly through shared dependencies or as dependencies of the same higher level packages. Based on the strength of correlations visible in direct dependencies for packages known to have similar roles, topic vectors that produce strong Pearson correlations for packages that perform related functions even if they had no predefined relationship in their definitions. Figure 4.7 shows the network map from Figure 3 rendered with edges for Pearson results above 0.3, generally considered the lower end of the "moderate" correlation range.[34] This map only shows new relationships produced via Pearson; relationships present in the original dependency

[34] The original layers of Figure 3 have been preserved in Figure 8, but horizontal positions have been optimized to show clustering. Because Layer 1 contains no source code, Figure 8's topmost layer corresponds to Layer 2 in Figure 3.

tree and not rendered. In total, 439 new edges were produced. Although topic modeling alone cannot tell us the nature of these new relationships, it is important to note that most of the new edges in Figure 4.7 occur in clusters. There is also a noticeable range of concurrency that is supportive of transparent design's strategies of concealment. Figure 4.8 shows the number of new relationships produced by layer. For each layer, the majority of new relationships formed are between packages on adjacent layers. Both Figures 4.7 and 4.8 both show that packages from the interface represented concepts from layers as low as 12, although considerably fewer after Layer 7. Figure 4.8 in particular shows that each layer has a representational range outside of which they share few topics in common with other packages. Programmers engaging in transparent design would thus seek to minimize those representation ranges.

Finally, Figure 4.9 abandons the dependency tree entirely in order to identify potential gatekeepers within Xfce. Figure 4.9 is a network map generated according to the strongest Pearson correlation between packages shows clustering of packages around similar topics. Creating links only through the highest Pearson result will not show the full range of direct relationships between packages; however, this network map produces a sense of how information might travel throughout the operating system. It's important to note that all of the packages that are positioned as gatekeepers—those that appear at intersections between branches of the map—are all located on Layer 5. Packages with topic vectors coded as user applications or GUI libraries, as well as those for internal/automated system management, are typically at the edge of the map. Abstract elements of the user interface and directly supporting libraries, in other words, are separated from system management packages. This separation on the network graph along with the positioning of Layer 5 packages as gatekeepers, further suggests that transparency is a targeted strategy. Figure 4.9 reflects the same separation between layers visible in Figure 5's

graph of similarities. The most radical of transformation within the Xfce occur across Layer 5. Figure 4.9 therefore suggests that while a dependency tree is useful for documenting functional links between packages, it is may be easier for researchers to understand transparency by not limiting comparisons to dependency relationships.

When considered within the context of GNU/Linux's history, these results suggest that transparent design and political transparency may not mutually exclusive. The presence of structure features that reveal the potential for transparency's strategies of obfuscation in Debian's dependency tree for Xfce certainly reflects the recent emphasis on usability within the GNU/Linux community. Nonetheless, the value that the GNU/Linux community places on forking and system customization means that Xfce, like other GUIs for the operating system, can be disabled or replaced. FLOSS, in other words, leverages modularity to permit users to move in and out of transparent environments, controlling the amount of system information they want to see but never denying them access to it outright. GNU/Linux's powerful command line interface is still present beneath it. In Figure 4.4, for example, those packages primarily represented by topics coded as command line utilities can be found near the bottom of the dependency tree. Using hotkeys to switch between virtual terminals, users quite literally are free to switch between the CLI and the GUI at anytime using hotkeys, avoiding any obfuscation or functional misrepresentation produced by the latter. Even if many computational processes are not represented within Xfce or other GNU/Linux GUIs, those processes may very well be accessible and directly configurable through text files. One major limitation of the tree produced by Debian's package definitions is that it assumes a basic system environment exists around. Libraries for wireless network access, which could themselves include interventions that unknowingly work against the users' wishes, are thus not considered here.

The limitations of a distant reading methodology leave open one important question: whether the appearance of transparent design in GNU/Linux's package network is the result of an intentional obfuscation or the cumulative effects of object-oriented programming's emphasis on modular encapsulation at a system-wide level. Given that Xfce is, by Stallman's definition, free software, it is somewhat surprising that it has features that reflect and would further support strategies of transparent design. Even though GNU/Linux is authored through a model of political transparency, it may very well be the case that the structural features expected of an obfuscated transparency are the due to the object-oriented paradigm. As the previous Chapter argues, transparent design and object-oriented programming both address problems of complexity by attempting to hide or otherwise filter it from our perspective. Nonetheless, the above case study suggests that transparency is more a targeted strategy than a blanket policy of computational suppression. Programmers who build transparent software, in other words, choose to exploit features of object-oriented programming to hide portions of the systems from users. The degree to which programmers hide or show computation to users is arbitrary, and digital media historians must interrogate the assumptions and intentions behind their decisions. GNU/Linux community maintains its operating system's reputation for customization and modification by acknowledging the way the object-oriented paradigm could potentially constrain the rights of its users just as Stallman acknowledged the political consequences of compilation. For digital humanists and historians of technology, the lesson here is not that computation inherently places limitations on its users but that it is vitally important for us to become more aware and self-reflective about the software we build, choose to use, and share our scholarship through. By uncritically accepting hardware and software platforms in our work, or choosing them primarily on technical grounds, we also accept any cultural assumptions their designers

make about the role computing should play in our society and what they understand to be the "proper" relationship between users, technology, and data.

Recently, the digital humanities has drawn considerable criticism for not engaging strongly with critical theory, at least by the standards of the home disciplines of its practitioners.[35] Digital humanists would do well here to draw on the methods and theories found in new media studies. Software is already in desperate need for strong, historically grounded scholarship given the frantic pace of its development. In some respects, it is true that digital humanists engage in the same problematic ahistoricism found in the sciences and engineering. In other words, while there is a rich body of technical knowledge accompanied with the skill to apply it to projects of impressive scale emerging from the digital humanities, there is not yet a strong framework for evaluating the cultural work of the tools and techniques of scholars in this field. Although this chapter does not presume to provide that framework, lessons from GNU/Linux's history nonetheless reveal some necessary first steps. A critical engagement with the theoretical assumptions tacitly present in software libraries is possible if and only if the source code for those libraries is accessible, in the very least for study as "open source" but ideally for modification as "free software." As with the rise of Stallman's critics in the GNU/Linux community, a push towards free software within the humanities will likely be met with arguments that digital tools need to be in some degree transparent in the sense that they

[35] See, for example, the essays that were part of the "Darkside of the Digital Humanities" panel at MLA 2013, as well as the conversations surrounding it on Twitter using the hashtag #s307. The essays begin here: http://www.c21uwm.com/2013/01/09/the-dark-side-of-the-digital-humanities-part-1/

should be readily usable without special training to prevent their insights from being monopolized by a small number of scholars—both to ensure that the digital humanities establishes a permanent place for itself and to prevent an influx of the same funding-based power dynamics that controls scientific and engineering research. These concerns are valid; however, the establishment of certain off the shelf, easy to use tools as standards should be understood as equivalent to declaring that one single school of critical theory is valid above all others. GNU/Linux and the FLOSS movements serve as reminders that arguments for technical superiority always come with often unacknowledged cultural consequences, and the digital humanities cannot afford to let them go unaddressed if it is to have a lasting impact on humanistic inquiry.

To this end, new media studies and the digital humanities should be understood as part of the same field rather than separate, albeit related, critical undertakings. As a disciplinary category, the digital humanities is already quickly becoming a big tent: grounded in its origins in the digitization projects of humanities computing and incorporating statistical analysis, pedagogical tools, and social networking discourses. The recent review of digital scholarship from Peter Lunenfeld, et al., *Digital_Humanities* (2012), is one of the first to also include new media studies under the label, rightly acknowledging that the term "digital humanities" also implies the cultural study of born digital texts. A stronger relationship between both of these fields of study will only help both. Consider, for example, Mark Sample's recent essay for Matthew K. Gold's *Debates in the Digital Humanities* (2012). In it, Sample argues that the digital humanities have "failed" because their methods are barred by copyright from examining the cultural periods from which their tools emerged. Yet as this project has shown, the analytical techniques used within the digital humanities can and should be applied to born digital texts.

Although such endeavors lamentably will leave Sample's desire to study Don Delillo with digital tools unfulfilled, the dialog they will produce between new media scholars and designers of distant reading software will serve as a strong foundation of a body of critical theory capable of highlighting cultural implications of technical decisions within digital methodologies.

Chapter 5 – A Material History of Mozilla

"In 2012 it made less and less sense to talk about 'the Internet,' 'the PC business,' 'telephones,' 'Silicon Valley,' or 'the media,' and much more sense to just study Google, Apple, Facebook, Amazon and Microsoft. These big five American vertically organized silos are re-making the world in their image."

—Bruce Sterling, from "State of the World 2013"

Pop-ups that gently remind users that updates are available from the appstore or that state more assertively that users should restart a program so that the latest version can be installed automatically are fast becoming common sights on personal computers. Today, developers can leverage an always online model of computer and device use to push updates to users' machines as soon as often as they wish. Some software, like the Mozilla Firefox browser, even have "rolling release," every-six-weeks development models that regularly push updates on a set schedule. Software, in other words, is subject to constant change, and outside of following changelogs—usually obtainable from a developer's website if users care to look—it is difficult to trace just how a piece of software changes unless there are alterations made to its user interface. Importantly, a way to map these inscriptive practices would allow us to look anew at theories of software's materiality by offering a chance to examine how the singular "applications" we interact with are formed through the iterative development of an assemblage of human written source code texts. While in the first three chapters of this project I examined the cultural history of personal computing software using paratextual sources—that is, texts written *about* software—in this chapter I propose that we can study the evolution of software

without relying solely on paratextual sources. In fact, as I argue, it's often the case that these paratexts do not fully represent software's changes. Although developers do document their activities using project management and versioning software, both must-haves for coordinating the labor of large teams, they do so primarily to keep track of task completion and to identify the moment a bug enters the codebase. Whereas I demonstrated in the previous chapter that topic modeling can help us map and interpret the object-oriented structures hidden by transparency, in this chapter I demonstrate that topic modeling can assist in tracing the evolution of software by showing us how topics that correspond to particular components grow or shrink over time.

Studying the history of software through its source code requires a reassessment of the medium's materiality. Although there is some consensus that software is an assemblage, there has been little consideration of software's iterative development model in digital media theory. Works by Robert Markley (1996), N. K. Hayles (1999; 2005), Jerome McGann (2001) and Matthew Kirschenbaum (2008), among others, have argued that far from the immaterial play of information in cyberpunk fiction, digital objects have form and structure, granting them material (or at least material-like) qualities. Recently, there has been some effort to move beyond theorizing about software's materiality and examine more closely how software is stored and rendered by computer systems.[1] At the same time, there are few accounts of the production of particular pieces of software or narratives of software's development over time aside from

[1] For example, see Kirschenbaum's forensic examination of a disk image of a game called *Mystery House* (1980) in *Mechanisms: New Media and the Forensic Imagination* (2008 or, Noah Wardrip-Fruin's discussion of the structure of various narrative engines in *Expressive Processing: Digital Fictions, Computer Games, and Software Studies* (2009).

insider narratives that give triumphant accounts of Silicon Valley's creative geniuses.[2] Unlike literary works or other cultural objects that are in some sense "complete" when released to the public, there are many examples of software that never really reach a state of completion. Software packages, including operating systems, web browsers, word processors, and graphic design suites, are updated continually for various purposes. A new version may be released, packaged, or sold as a distinct, finished product even though it is built upon the same continually evolving codebase shared by previous versions.

Digital media theory has yet to account for iterative development models. Instead, studies of digital culture often implicitly assumes that software, or other electronic texts, have fixed, authoritative versions. Digital media scholars can frame important theoretical questions without the need to examine software's production when working with a single version of software from an aesthetically driven genres of software—such as video games or electronic literature. However, this focus on single versions of software leaves us unable to articulate how the constant parade of updates, patches, and security fixes allows developers to exert a long lasting influence on user behavior. As Chapter 2 argues, transparent design continually shifts our attention to the present and a continual re-imagining of a future waiting just ahead, discouraging

[2] There are only two studies comparable to this chapter currently in print. The earliest is Kirschenbaum's account of the various "editions" of Michael Joyce's electronic novel, *afternoon*, in his 2002 essay "Editing the Interface: Textual Studies and First Generation Electronic Objects." The other is Ian Bogost and Nick Montfort's *Racing the Beam* (2009), which traces the history of the Atari 2600's production through a review of documentation, popular periodicals, reverse engineered system specifications, and interviews.

us from considering the cultural history of new technologies. Without the ability to recover this history, we will never be able to understand the full extent of the control that developers have over their software even after it has been installed on users' personal machines.

This study contributes to expanding our understanding of software's materiality by proposing that the epistemological problems posed by software's iterative development models can be accounted for through a distant reading of its source code as a versioned corpus. The free and open source software (FLOSS) community's tendency to archive the entire development history of packages makes it possible to examine software's history directly, but poses new methodological challenges. As Chapter 3 argues, modern application software written in the object-oriented paradigm resists close reading. Not only does object-oriented programming support software on a scale beyond human comprehension, but it manages that complexity by encouraging readers and writers of source code to adapt functional, yet incomplete perspectives on large bodies of source code. Object-oriented programming, in short, produces a complex, intertextual body of source code: a massive archive even for a single version of software. Complex bibliographic problems are not new to the humanities, nor to digital media studies. Kirschenbaum's 2002 essay, "Editing the Interface: Textual Studies and First Generation Electronic Objects," argues that digital objects have a "material complexity" that requires "critical approaches capable of accounting for such phenomena as multiple versions and releases, data standards, platforms, and file formats, all of which contribute to the textual composition of electronic objects. The intellectual precedents for such an approach are clearly to be found in textual criticism and bibliography" (43). Yet the sort of manual editing that Kirschenbaum demonstrates with electronic literature would be difficult to perform on large bodies of source code, even for a small handful of versions. As I argue in this chapter, this problem of scale can

be addressed using a workflow that incorporates latent Dirichlet allocation topic modelling tools and accounts for the amount of source code repetition across versions of software resulting from iterative inscription.

Following a discussion of the viability of topic models for software history, I then move into this chapter's case study. For this chapter, I've chosen to examine the Mozilla Firefox browser because it is one of the few pieces of software to have existed, in some form, as both open and closed source. The source code to Mozilla Firefox can be traced with few interruptions back to 1998: the year that Netscape publically released the source code to its browser before being bought out by America Online, signaling the end of the "browser wars." Mozilla Firefox, in some form, participated in the early visions of information culture and bears within its versioned history contributions and responses to the commercialization of the Internet. After first providing a context for the case study by tracing the history of Netscape as closed source, I then turn my attention to Mozilla Firefox's source code. My case study covers examines 60 versions of Mozilla Firefox's source code which together cover a 15 year span (1998-2013), hereafter referred to as the "Mozilla Corpus."[3] Table 5.1 is an index of the corpus, providing basic

[3] A note on version selection: The corpus includes all released versions of Mozilla Communicator and Mozilla Application Suite up to the 1.0 release of Suite. From this point forward, very small updates to each version were not included. Although bug fixes and security patches do reflect a continually changing body of code, it's assumed that these small updates would be still be part of the codebase at the next major release. Without this exception, the number of versions in the Mozilla Corpus would have at least doubled. Additionally, because Mozilla Suite and Firefox overlapped in development, Firefox does not enter the corpus until its

information about each version as well as the label number used in all Figures to follow. In addition to its implications for the study of transparency, software history, and textmining in the digital humanities, this chapter's case study of Mozilla Firefox also suggests that the two languages that comprise most websites—Hypertext Markup Language and JavaScript—are changing to match the closed source and object-oriented models of development found in consumer software. My topic models show that Mozilla Corpus has undergone an extensive structural transformation over the course of 15 years that essentially recast the role of HTML and JavaScript from document rendering languages into application interface languages. In this respect, the Mozilla Corpus serves as an excellent example of the conceptual disconnect between a transparent interface and an application's core libraries.

I. Methods: On the Viability of Topic Modeling for Software History

Theoretic models of software that incorporate some notion of bibliography impoliticly frame the person computer itself as a media ecology and examine the movement of signifiers through it. For example, Hayles responds to Kirschenbaum's call to "take electronic texts seriously as *texts*" in *My Mother Was a Computer* by revisiting her concept of the "flickering signifier." Hayles defines "digital text" by describing the way the signifiers in machine code are produced across software's vertical layers are made visible on computer screens as natural

---

first 1.0 release. Because the earliest versions of Firefox, originally dubbed "Phoenix," were not publically released, its individual timeline would be incomplete. Second, development on Mozilla Suite was halted shortly before the release of Firefox 1.0. Fortunately, as discussed below, the two applications shared many core libraries, so the transition was not as abrupt as that between Communicator and Suite.

language signifiers through a series of algorithmic transformations (97-104). Her notion of a complex bibliographic history is the trail of interpretations or translations that occur within machinic memory between the signifiers visible on screen and the binary code stored in a computer's file system. In other words, each subsequent representation of the natural language text, from the bare metal representation on magnetic discs to the graphemes on screen is part of that bibliography.[4] Hayles is not alone in this formulation, nor is it necessarily limited to digital media theory. Lawrence Lessig, a long-time legal activist for copyright reform, has argued that copyright law cannot apply to digital texts because every act of access produces a bibliographic trail of copies. In fact, some of the latest legal arguments in favor of opening copyrighted works to the digital humanities invoke a similar model and propose that permitting access just before the bibliographic trail reaches a human readable copy would allow for a form of "non-consumptive" or "non-expressive" access.[5] But software, understood as packages of algorithmic instructions, has a complex bibliographic history apart from what appears on screen. If we consider that software is "written" as a set of instructions in a programming language and treat this "source code" as a text, then software's iterative development and versioned releases result in bibliographic histories that very quickly begin to rival the detailed histories produced by critical editors working in textual theory. Table 5.1 shows the number of tokens, or words, in each version after commonly repeated types were removed. Even when reduced in this manner,

---

[4] Kirschenbaum himself takes up the challenge of examining this chain of signifiers in his forensic study of a disk for the game "Mystery House" in *Mechanisms* (2008).

[5] See Matthew Sag's "Orphan Works as Grist for the Data Mill" forthcoming from the *Berkeley Technology Law Journal*. http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2038889

the smallest versions would still span the length of multiple novels. Comparing even a few by hand would in itself be a large project.

Editors working in literary history are no stranger to weighing the cultural influence of particular versions of texts or investigating the intentions behind changes across them; however, even if the changes between editions of a literary text are well documented in annotated, critical editions, the problem of negotiating between competing versions of an object of study complicates matters. In the case of software versions, each is in some sense equally legitimate at its moment of release, representing the intentions or vision of its development team at a given point in time. As Jerome McGann notes in *A Critique of Modern Textual Criticism* (1983), there are a number of problems with establishing textual authority based on the reconstruction, or even explicitly discovery, of an author's intentions. McGann calls on editors to (re)construct scholarly editions through a strong reflection on how the goals of their edition fit within a work's historical context, and his comments regarding the historically constructed nature of literary works are relevant to software as well. Traditionally, a romantic view of the author as the singular source of literary production led editors to search for and remove "structural contaminants on an original and autonomous authority" that resulted from the material transmission of a work (103). Reconstructing an author's intended copy-text is difficult, a problem that becomes more complicated when there are multiple editors and publishers involved in transmission. McGann thus argues that we must acknowledge that literary authority is dispersed as a condition of production because all works are necessarily "social projects which seek to adapt and modify themselves circumstantially" (102). In other words, a text's authority rests neither solely in its author, nor in the editor who receives it, but also in its transmission through time. McGann discovered later in his career that digital tools, in his case hypertext, can bring older texts into a

new medium, allowing editors to document for their readers the historically constructed nature of literary works in ways not possible in printed critical editions; however, McGann's problem of versioning involved a small number of texts with explicitly aesthetic dimension. His use of hypertext to produce interactive, critical editions, such as the Rossetti Archive, that allow scholars to study the historical transmission of texts only supports close reading.

In the case of software's textual history, the problem of competing, yet authentic versions posed by iterative development models is compounded both by archival scale and the complexity of software's structure. As I argued in Chapter 3, software studies by and large does not directly address the way that object-oriented principles shape software into an abstract network of encapsulated components. Nonetheless, there is some degree of consensus within digital media theory that software's materiality is an emergent property resulting from the interaction of its various components. Hayles, for example, explains in *My Mother Was a Computer* that users perceive software on-screen as a singular digital object even though it is produced from "data files, programs that call and process the files, hardware functionalities that interpret or compile the programs, and so on. It takes all of these together to produce the electronic text. Omit any one of them, and the text literally cannot be produced" (101). Friedrich Kittler (1999), Lev Manovich (2001), Wendy Chun (2005, 2011), and Matthew Fuller (2005) have offered similar descriptions of software's structure. It is true that software is functionally an assemblage, drawing its various components together and transcoding them into cultural forms recognizable to users. Yet, as Fuller notes, breaking down assemblages into components parts can quickly produce a sprawling network of components that is difficult, if not impossible, to pin down:

> Multiplicity is induced by two processes: the instantiation of particular compositional elements and the establishment of transversal relations between

> them. The media ecology is synthesized by the broke-up combination of parts. Each compositional fragment, each item on the list—while being under the effect of certain grammatical schema nameable as an object or a whole thing. . .—opens into other permutational fields. Each part, then, forms an axis to which this shifting patchwork can be connected (16).

Even single applications are structured as assemblages, especially if they are written in an object-oriented language. Although there is a limit to their decomposition, the number of objects and classes, each developed by different teams of programmers, that comprise a complex piece of software, like a web browser, makes it difficult for a single person to track the evolution of each through time. As the previous chapter demonstrates, software's sprawling assemblages can be mapped and interrogated using topic modeling. Topic modeling source code can synthesize functional descriptions of software components, providing a conceptual map for interpretation. Yet the GNU/Linux case study involved a synchronic corpus of varied applications and libraries that represented a snapshot of operating system architecture. Because each software package was within the context of the corpus a unique document, the GNU/Linux case study shared many resemblances to the corpora used for literary topic modeling. In order to account for the amount of source code carried over between versions in an iterative development model, the Mozilla Corpus must be done in a way that recognizes the effect that repetition has on topic modeling's probabilistic algorithm.

Today, most topic models are produced using some implementation of the latent Dirichlet allocation (LDA) algorithm first proposed by David Blei, Andrew Ng, and Michael Jordan

(2003).[6] As Blei, Ng, and Jordan note, the advantage that topic modelling provides over other methods of document comparison, such as the commonly used tf-idf scheme, is that it can reveal patterns of intra- and inter-document statistical structure. Unlike tf-idf, a weighted word counting algorithm, topic modeling allows for equivocal meanings, treating each instance of a word as distinct yet potentially related to other instances. LDA assumes that all documents in a collection are comprised of words (or some other form of token) representing a finite group of topics. The algorithm treats documents as if they are constructed as a mixture of topics. These "topics" are defined as statistically distinct distributions of words, meaning that they randomly populate with words according to certain probabilities with no regard for sequence. The advantage of LDA over other topic modeling algorithms, like the probabilistic latent semantic indexing (pLSI) algorithm described by Thomas Hoffmann (1999), is that it is not supervised, meaning it does not require a set of training documents that represent desired patterns. Instead, LDA assumes that topics are present in every collection, and operates with the understanding that with enough examples of each topic throughout the corpus, testing and adjusting probabilities of distribution patterns will eventually isolate each topic's pattern.

LDA begins by first randomly assigning topics to each token in the corpus. It then checks both the number of words assigned to each topic in a document as well as the proportion of topics assigned to each instance of a particular word across all documents. Using this information, it then calculates both the probability that each instance of a particular word will be assigned to each topic and the probability that each topic will be a part a document's topic

[6] Unless otherwise noted, this following description of LDA is summarized from Blei, Ng, and Jordan's article.

mixture. LDA then revises the topic assignment of each word in the entire corpus using these newly calculated probabilities and begins the process again. Whereas pLSI's training documents suggest a correct distribution of topics across documents, LDA assumes that distributions are weighted according to a "hidden" variable. The goal of LDA is not to assign words to topics so that topics in the documents under analysis will have a similar distribution to those in a training set but to revise distributions over and over again in an effort to discover the hidden weights of each topic. Although this hidden weight may not explicitly exist, LDA assumes that in linguistic practice people implicitly weigh certain words as more important than others when discussing a given subject. Given enough iterations, LDA's topic mixtures and assignments will eventually stabilize around these imagined weights.

LDA will always produce statistically distinct topics; however, using unsupervised algorithms to filter meaningful patterns out of linguistic data poses epistemological problems. As Alain Liu (2013) notes in a consideration of unsupervised machine learning algorithms that it is "not clear epistemologically, cognitively, or socially how human begins can take a signal discovered by machine and develop an interpretation leading to a humanly understandable concept" (414). According to Liu, there are a great number of assumptions behind the idea that an algorithm like LDA "works" which aren't necessarily acknowledged by researchers in the digital humanities. Like LDA itself, anyone who applies the algorithm also assumes that there are pre-existing "topics" within a corpus. For such a signal to emerge from the statistical noise of a large corpus, Liu continues, that signal must contain "a coeval conceptual origin that is knowable in principle because, at a minimum, the human interpreter *has* known its form or position (the slot or approximate locus in the semantic system where its meaning, or at least its membership in the system, is expected to come clear)" (414). In this respect, Liu is less critical

of the idea that these algorithms are useful for humanistic inquiry than with the idea that these algorithms, in being "unsupervised" in the sense that they can provide an objective perspective or one that is otherwise capable of discovering an essential truth in human language that humans themselves cannot see. In this respect, Liu's concerns resemble the same ones that Brian Rotman saw for mathematicians during the early days of computing, namely the question of whether a machine can understand mathematics better than its human creators.

For Liu, the epistemological problem posed by unsupervised algorithms must be addressed by acknowledging the limitations of mechanic perspectives and finding ways to account for them that draw on humanistic frameworks of knowledge. As an example, Liu recounts Ryan Heuser and Long Le-Khac's decision to incorporate information from the *Historical Thesaurus of the Oxford English Dictionary* in order to improve their semantic clustering algorithm. According to Liu, Heuser and Le-Khac's project keeps interpretation at the forefront of their computational efforts through a series of adjustments to their algorithms made to reflect their understanding of their corpus' historical context. Although addressing Liu's concerns for literary studies are beyond the scope of this essay, I agree that any use of algorithmic tools by humanists should be prefaced with responses to the questions that Liu raises. In practice, the potential for topic models to represent concepts in a corpus is largely dependent both on how researchers prepare their corpora and how they integrate the LDA algorithm into their workflow

First and foremost, using LDA to study a bibliographic history of a text—be in the source code to a piece of software or multiple editions of a play—requires a workflow that accounts for the textual duplication produced via iterative inscription. In the digital humanities, topic modeling is often used to synthesize historical trends in large corpora. As Matthew Jockers

demonstrates in *Macroanalysis: Digital Methods and Literary History* (2013), once topic models are produced, the topic memberships of documents can be grouped by publication date. Memberships can then be plotted chronologically, allowing the topic models to show the prevalence of particular concepts within the corpus over time and interpreted using contextual knowledge or closer investigation of individual texts. A key difference between the sort of literary corpora that Jockers uses and a versioned corpora of source code is that the former have very little, if any, duplicate texts. Because LDA functions by examining the distribution of topics assignments within and across documents, duplicate texts can skew the LDA's probabilities by giving more weight to the topic mixtures within the duplicates. With a large enough corpus, a small percentage of duplicates might have little noticeable effect; however, in a versioned corpus, large passages of text are carried forward from version to version, meaning that the topics generated by LDA would reflect primarily the mixtures of words found in the most static portions of text, making it an ineffective tool for studying patterns of textual change.

Topic modeling a corpus with this much duplication thus requires a workflow that prevents duplication from entering into LDA's calculations but which also acknowledges those components of the software that carry over between versions. Stephen W. Thomas, et al. (2011) argue that LDA's tendency to weigh its models towards duplicated passages can be avoided if only the changed passages between each version are prepared for topic modeling. Although there is a considerable body of work from engineers applying distant reading techniques to source code, only Thomas' article accounts for the software's iterative inscription. Thomas proposes that the duplication in software's complex bibliographic history can be accounted for in topic modeling by wrapping the LDA algorithm in a pre- and post-processing scheme that tracks changes between versions. Following Thomas' recommendations, I wrote a Python interface for

the Unix "diff" utility. When diff is run on two directories—with one designed as original and the other as copy—it compares each file with matching names in both directories, representing each pair in terms of the portions added and removed between versions. Following Thomas' description of the workflow, my Python interface parses the output of the diff query and creates a pair of δ-add and a δ-remove documents for each file containing C or C++ code. Because LDA is not affected by the sequence of words in a document, it does not matter whether changes to different parts of a file are positioned closely together in the δ-documents. In the event of whole files added or removed between versions, the entire files are treated as δ-documents. Once this step has been performed for every file, the δ-documents are then pre-processed as normal.[7]

Following LDA, I had to cumulatively sum the resulting topic models in order to assemble representations of the changes to each version. The results of the LDA were a series of δ-add and δ-remove document, up to several hundred for each version of Mozilla Firefox. To calculate the topic membership change for each version, I summed all of the δ-add and δ-remove documents for each version, resulting in 120 vectors. Finally, I prepared representations of each version of the Mozilla Firefox codebase by cumulatively summing the δ-add and δ-remove (i.e., Version 4 is the sum of the δ-vectors for Versions 1-4). Thomas concedes that the diff model can on occasion be overly sensitive to changes in the corpus due to LDA's probabilistic methods. I noticed that when the net change between versions was negative, that some topics often Topics produced using the diff-model initially proved unstable when the majority of a version's files

[7] See Chapter 4 for a complete description of my source code pre-processing procedures. The only difference in pre-processing is that this case study uses an 80% common token threshold to filter context specific stopwords.

were treated as δ-remove documents, generally at the points when the Netscape codebase was abandoned during the transition to Mozilla Suite and when development switched from Mozilla Suite to Mozilla Firefox.[8] To account for these non-iterative transitions, the cumulative summing of the virtual vectors was restarted, with topic memberships reset to 0 and all files in the first version after the restart modelled in full as δ-add documents.

As with many topic modeling projects, I had to determine an appropriate number of topics for LDA to generate through a process of trial and error. In the remainder of this section, I share a few observations about how topic models of different sizes affects its representation of source code. Although LDA does not require any sort of training data to produce a topic model, it does require that the number of topics be set in advance. Unfortunately, there is no systematic method for determining a good size for a topic model. As Jockers explains:

> There is neither consensus nor conventional wisdom regarding a perfect number of topics to extract, but it can be said that the "sweet spot" is largely dependent upon and determined by the scope of the corpus, the diversity of the corpus, and the level of granularity one is seeking in terms of topic interpretability and coherence. . . .[T]he "interpretability and coherence" of a topic mean specifically the ease with which a human being can look at and identify (that is, "name" or "characterize") a topic from the word list that the model produces. Setting the

[8] Even though it's clear from documentation that Mozilla Suite and Mozilla Firefox share core libraries, diff expects new versions of files to be organized in the same way as previous files. Changes in organization caused all Mozilla Suite files to be treated as δ-removes, producing erratic vectors at the point of transition.

> number of topics too high may result in topics lacking enough contextual markers to provide a clear sense of how the topic is being expressed in the text; setting the number too low may result in topics of such a general nature that they tend to occur throughout the entire corpus.

In the case of a source code corpus, the size of a topic model generally determines the level of abstraction the resulting topics will represent. In the previous chapter, I found that using a number of topics roughly equivalent to the number of documents in a corpus comprised of different software packages produces topics that correspond to the particular function of each package. Shrinking the number of topics forced the LDA algorithm to produce topics that represented more abstract, general functions: a “graphics library” topic rather than a “subcomponent of a graphics suite dedicated to three-dimensional shading” topic. This size allowed for an easy classification of each package, enriching the dependency tree to show how functions are distributed throughout the network of libraries and applications necessary to display a desktop environment in Linux.

A similar pattern of abstraction occurs when a corpus constructed from multiple versions of a single application is topic modeled. Rather than producing topics that match entire libraries, a large model can produce topics that match particular implementations of classes and primitive objects whereas a smaller model produces topics that reflect the abstract classes that serve as the basis for particular implementations. In an object-oriented language like C++ or Java, implementations produced through polymorphism are already structurally related through the abstract class they inherit properties from. A small model size will produce an abstract perspective, collapsing related implementations into a single topic. Figure 5.1 shows three topic

vector graphs taken from three different topic model sizes of the Mozilla corpus.[9] These graphs represent portions of the XPConnect library, which serves as an internal interface between Mozilla's JavaScript engine—the component that executes JavaScript commands—and Cross Platform Component Object Model (XPCOM), which serves an abstract interface to system-specific processes like memory management, file system, internal messaging libraries, etc. The drop in each graph at label 29 reflects a decision to revise an older implementation of an XPConnect that required two intermediary interpreting libraries, called "wrappers," before JavaScript commands could reach the necessary modules within XPCOM.[10] This revision made

[9] A note on how to read all topic graphs in this chapter: The red line represents each topic as the total number of tokens assigned to that topic, and the blue line represents each topic as a membership value, or the percentage of all tokens in the corpus assigned to that topic. The first vertical solid line at label 7 indicates the first version of Mozilla Suite, and the second vertical solid line at label 37 includes the first version of Mozilla Firefox. The first dotted vertical line at label 29 is Mozilla Suite 1.0, and the second dotted vertical line at label 44 marks Mozilla's shift to an accelerated "every six weeks" rolling release model for Firefox. Full color galleries can be found at http://mblack.us/chap5

[10] Unfortunately, this decision was not documented in the "What's New" change log that accompanied each public announcement of a new version. This account was pieced together from a review of white papers accessed through the Internet Archive's cached history of the Mozilla Foundation's website and bug report logs that data back to the open source release of Netscape Communicator. Nonetheless, it serves as a good example of the potential for topic modeling to highlight development decisions that were not well documented by developers.

the transfer of data from Mozilla's JavaScript components to XPConnect more direct, significantly reducing the amount of source code devoted to these operations. That the sudden drop occurs at the release of Mozilla Suite 1.0 (label 29) is purely coincidental. Graphs with smaller numbers of topics also show an increase in token assignments after the transition into the Firefox era. The difference between the smaller and the larger models is due to the level of abstraction produced by the number of topics. In smaller models, LDA is forced to represent XPConnect in a fewer number of topics, making no distinction between the removed wrappers, the wrappers remaining after the change, and later ones added as part of Mozilla's expanding cross-platform strategy for Firefox. A larger number of topics, in other words, makes individual components more visible; however, it makes it more difficult to make connections between related or early and later versions of components. For the results discussed below, I decided to use a model size of 60 topics because it allowed for sufficient distinction between component types without too much overlap between the key terms generated for each topic.

II. Results: Exploring the Materiality of the Mozilla Corpus

The Mozilla Corpus serves as an excellent example of how much code duplication is present in a versioned history of software. Table 5.1 shows the net change in token count between each version, and Figure 5.1 shows the percentage of change between versions for Mozilla Suite and Mozilla Firefox. Across sixty versions, there is only one point at which the code base was entirely set aside and development rebooted. Prior to Netscape's release of the source code for their Communicator Suite in 1998, the company was developing two parallel

---

See: http://web.archive.org/web/20021002150509/http://www.mozilla.org/scriptable/js-components-status.html and https://bugzilla.mozilla.org/show_bug.cgi?id=71483

versions of their browser. Both were eventually abandoned, but parts of version 5.0 were released as open source. Version 5.0, written in C++, was a continuation of the code base that the Netscape team had been working on since the browser's beginnings as a re-imagining of NCSA Mosaic, the first graphical web browser, developed in part by Netscape's co-founder Marc Andreessen.[11] The second browser, version 6.0, was to be a complete port of Navigator to the Java programming language, dubbed "Jazilla" by the press. Andreessen intended for Jazilla to be the future of the company, using the cross-platform compatibility afforded by Java's virtual machine structure to deliver fully on Netscape's promise of producing a universal browser that behaved and rendered the Internet identically on any machine. In the end, however, Jazilla was abandoned as Netscape struggled to compete against Microsoft. Version 5.0 thus represented an attempt to keep up with Microsoft in the race to build new browser features that would hold users' attention until the release of Netscape's next generation Java browser.[12]

Although Netscape's decision to release the source code to Communicator was framed by as a grand gesture of support for the growing open source software community, accounts of the company's situation in the marketplace suggest that the release of Communicator as open source was more or less a last ditch effort to see development of Netscape's technology continue as the

---

[11] Despite some similarities, NCSA Mosaic and Netscape Navigator do not share any direct ancestry. The Mosaic license was eventually acquired by Microsoft, who used Spyglass' implementation of the browser as the foundation for Internet Explorer.

[12] See pages 188-198 of Michael A. Cusumano and David B. Yoffie's *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft* (1999) for interviews with former employees about the plans and decisions behind these two versions.

company faced a buyout by America Online.[13] Before open-sourcing Version 5.0, however, Netscape had to remove large portions of code that it had licensed from its partners. Thus, the first open source ancestor to Mozilla Firefox was released in a hobbled, yet still functional, form. Following the buyout by America Online, Netscape abandoned the old source code and decided to rebuild its Communicator Suite from the ground up, and on December 11, 1998, the earliest version of what would come to be known as the Mozilla Application Suite (or "Mozilla Suite" for short) was posted to the Mozilla Foundation's ftp server.

Although technically speaking, development was also ceased on Mozilla Suite in 2004 in favor of a more stream-lined application that would be released simply as a browser and not as a collection of Internet tools, Mozilla Firefox was based on the same core components as Mozilla Suite.[14] In a web browser like Mozilla Firefox, the component that renders information written in HTML/CSS or other "web languages" is known as a "layout engine."[15] Netscape had begun working on Mozilla's layout engine, Gecko, prior to 1998 and had intended it to succeed the

---

[13] In addition to Cusumano and Yoffie's book, see also Thomas Hugh's essay "Protocols for Profit: Web and E-mail Technologies as Product and Infrastructure" in *The Internet and American Business*, eds. William Aspray and Paul E. Ceruzzi (2008).

[14] For more on Firefox as a "streamlined" revision of Mozilla Suite, see: http://web.archive.org/web/20110623034401/http://weblogs.mozillazine.org/ben/archives/009698.html

[15] For more on layout engines or Gecko in particular, see Mozilla's Gecko FAQ at: https://developer.mozilla.org/en-US/docs/Gecko/FAQ?redirectlocale=en-US&redirectslug=Gecko_FAQ

engine they had been modifying and adding to since their first release of Navigator; however, in 1998, Gecko was released as an open source library along with what remained of Communicator and not incorporated into any browser until development began on Mozilla Suite. Although it is true that some components of Mozilla Suite were abandoned during the transition to Mozilla Firefox, Gecko and its supporting modules were carried over. A true break from Mozilla Suite does not appear to occur until label 40, Version 3.0 of Mozilla Firefox. As the graphs in Figure 5.2 show, the total change to the C++ codebase are 85% of the size of the previous version, with a net reduction in size of 21.5%. There are also noticeable changes in the trends of individual topics that occur at this point. As Figure 5.3 shows, non-browsing components like e-mail that were a part of the bundled applications model used by Communicator and Mozilla Suite are not actually removed from the C++ codebase until version 40.[16] Despite claims that Firefox was to be a clean break from Mozilla Suite, the two are more directly related for a brief period following Firefox's version 1.0 release than Netscape Communicator and Mozilla Suite. In short, the repetition in the Mozilla Corpus is extensive, reflecting its production through a model of iterative inscription, and making it unsuitable for the same topic modeling workflows in use by digital humanists.

[16] This break coincides with an announcement by Mitchell Baker, chairperson of the Mozilla Foundation's board, claiming that increased resources would be devoted on Thunderbird, an e-mail application built on Gecko. The removal of e-mail code from the Mozilla Corpus at this point suggests that the move was as much about streamlining Firefox as it was about improving Thunderbird. These algorithms were likely rewritten in JavaScript rather than added into Thunderbird's C++ codebase. See below for more on the structural changes in Firefox/Gecko.

One major trend that stands out among the Mozilla corpus' topic vectors is a pair of sharp declines in topics at either end of the versioned timeline related to rendering the browser's GUI—menus, windows, tabs, icons, button. Figure 5.4 shows three topics exhibiting this trend. Even discounting this trend, the huge spike in topic 56 during the Communicator era stands out compared to the others in the model. The vast majority of topic vectors show an upward trend moving forward in time, plateauing either in the Mozilla Suite era or near the beginning of the Firefox era, if not continuing to rise into the present. Topic 56, on the other hand, shows not only a spike off the scale of the graph in terms of both token counts and version membership, but also exhibits a downward trend throughout the version timeline. According to the LDA results, topic 56 is defined by distribution of the terms (in order of probability): item, window, menu, view, drag, rect, command, button, wnd, cmd, select, bar, dlg, edit, widget, frame, icon, messag, control, hwnd, drop, dialog, handl, and win. Although some of these terms are shared with other topics, the overall function they appear to represent is the rendering of interface objects. Given the comparatively limited graphics capabilities of personal computers at the earliest parts of the timeline, it is curious that so much of the code-base would be dedicated to graphics libraries during the earliest periods of the Mozilla corpus. As Figure 5.6 shows, there are a number of topics related to graphics that reflect the assumption that rendering capabilities would rise alongside increases in consumer computing power.

Histories of the browser wars highlight Netscape's corporate strategy of interposing the browser as a new platform for business that would run on all major operating systems and host its own set of applications built using web languages. Their goal, in other words, was to make users dependent on Netscape Communicator for their daily tasks, regardless of the operating system they used. During the conflict between Apple and IBM discussed in Chapter 2, Microsoft

quietly carried out a similar strategy with hardware vendors, licensing and spreading its own implementation of the Disk Operating System to everyone producing clones of IBM's PC. Although there were only three major competing operating system platforms in 1994 when Netscape released version 1.0 of its Navigator browser, the company promised users that web pages would look the same across all of them.[17] Representing it's browser as universal was not so much a move to attract users—according to Cusumano and Yoffe, Netscape was already used on 90% of Internet connected personal computers by 1996—as it was a way to sell its own web server and developer tools. Andreessen typically used his "TechVision" blog to promote new and upcoming tools for the company's corporate customers:

> Netscape ONE is doing for application development what HTML did for information sharing. Why do I say that? HTML changed the way people could share information by leveraging the ubiquity and platform-independence of the Internet. Netscape ONE changes the way developers can create applications by allowing fully functional applications to be built in the same Internet framework. This framework provides the ability to write software once for multiple platforms, reach more people than ever before, distribute and deploy it for trivial costs, and enjoy the security of universally supported open standards. Like HTML, Netscape ONE works for the entire Internet. Also like HTML, it doesn't assume that you and everyone you want to communicate with use a

[17] This idea has since become a key part of Internet technology. For more on how an experience of "seamlessness" shapes information technology, see Alexander Galloway's *Protocol: How Control Exists After Decentralization* (2004).

particular operating system. It isn't Netscape's platform, it's the Internet's platform.[18] Although these comments serve more as advertising than as useful information about the company's plans, it's worth nothing that Andreessen frequently associates advancements in Netscape's products to changes in the Internet itself. In fact, Andreessen tried to cultivate the idea that Netscape would leverage its dominance of browser software to introduce sweeping changes to personal computing as Internet access became more widely available. Bob Metcalfe reported in 1995 that Netscape's goal was to build more and more features into its browser until it could function like an operating system, hosting all of users' computational tasks and thereby turning Microsoft's Windows and Apple's MacOS into "a mundane collection of not entirely debugged device drivers" (111). While toolkits like ONE or the Netscape OS left little impression on history, they hint at the future direction of Netscape's, and later Mozilla's, browser technology.

Netscape's push to position its browser suite as a universal interface helps to contextualize the trends visible in Figure 5.3 in two ways. First, the large spike during the Communicator period likely reflects the complexities of generating complete user-interfaces with similar look-and-feel across three different operating systems. Figure 5.6 shows unscaled versions of both the token count and the percentage of code-base occupied by topic 56 over time. Promising a universal browser, in other words, meant in practice that 30% of the code-base was dedicated to accounting for the peculiarities of the platforms Netscape was executed on top of.

---

[18] See:

http://web.archive.org/web/19970227011735/http://www15.netscape.com/comprod/columns/techvision/one.html

At its peak, those components represented by topic 56 accounted for 267,799 tokens with the next highest peak in a different topic vector being 151,653. This second peak is also during the Communicator era in a topic representing e-mail management. The third highest (and first outside of Communicator) is 128,982 during the Firefox era in topic 40, included in Figure 5.5, which represents HTML5 graphics elements. Topic 56's peak is especially important to note given that occurs in the era with the smallest size C++ codebase, peaking at 1/2 the size of the entirety of the Mozilla Suite codebase and 1/5 the size of Firefox's (see Table 5.1). Although Netscape's design plans during the closed source period were never made public, it is a safe assumption that this peak represents cross platform components written in Java that could not be released as open source because Sun Microsystems would not allow Netscape to release its browser with built-in support for Java.[19]

Nonetheless, the fact that the trends in topics 22, 27, and 56 begin to resemble one another despite the wild rise and fall of 56 in early versions suggests that Mozilla carried on Netscape's plan to become a universal interface. According to the published proposal notes from a meeting with Netscape's management under America Online, developers began working on the Cross Platform Toolkit, later renamed the Cross Platform Front End (XPFE), that would "leverage Gecko to maximize performance [and] minimize [the] effort and overhead" required to

[19] According to an interview with Aleks Toctic, one of Netscape's developers, conducted by Cusumano and Yoffe, there were internal versions that tried to use Java to implement cross-platform, uniform user interface components, but these were abandoned sometime in 1998, the same year as the open source release (178). As a result, all user interface components had to be re-written in C++.

manage the user-interface.[20] In other words, developers wanted to design the Mozilla Suite's user interface components using the same layout engine that rendered web content. Archived documentation for XPFE shows that the framework acts as an intermediary between user interface elements rendered on screen and the system-specific libraries present within an operating system.[21] Rather than explicitly encoding every element of the interface in C++, XPFE renders the interface by interpreting instructions written in XML, HTML/CSS, and Javascript. In addition to reducing significantly the amount of source code needed to manage the user interface, it would also make it more flexible: user interfaces could be "replaced" by downloading new instruction files or added to with downloadable components. This observation is supported by the recovery of topic 56 after its sharp drops end in 1999 (at label 15), suggesting that the components it represents still managed the relationships between elements of the user interface but no longer defined the behavior of those elements. While most of topic 22's terms are all explicitly related to web page components, the term "chrome" appears to stand out; historically, however, "chrome" is the term Mozilla has used internally to refer to the user-interface rendered from XUL and JavaScript documents, marking the topic as part of XPFE. Topic 27 therefore likely represents those components resting between XPFE and the operating system specific libraries like window managers or the event listeners that track mouse and keyboard input. Topic 27, for instance, shares many key terms with topic 16, such as "key" and "mouse" (see Figure 5.8). One important difference between the two is that topic 27 includes "gtk," which stands for the GNOME Toolkit, a graphics library commonly used within GNU/Linux, whereas topic 16

[20] Full notes can be found here: http://www-archive.mozilla.org/xpfe/ExecReview.html

[21] http://www-archive.mozilla.org/xpfe/orig/xpfe.html

includes “hwnd,” an ID number used in Windows to track individual instances of window objects. Yet these vectors differ in the rise and fall at the end of the vectors for topics 16 and 27, respectively. Presumably this discrepancy represents the different amounts of code necessary to manage relationships between the browser and graphics libraries in Windows or GNU/Linux, respectively.

Mozilla’s decision to remove explicit definitions of interface elements in C++ and instead interpret them from XML, HTML, and JavaScript files represents the beginning of a profound change to the browser’s materiality. As Figure 5.8 shows, XPFE eventually resulted in a sharp, but temporary, increase in the amount of XML bundled in with Mozilla Suite’s C++ code base; however, there is also a steady increase in the amount of JavaScript code included beginning at the same time as XPFE’s introduction in 1999 (label 15).[22] At label 41, version 3.5 of Mozilla Firefox, it rises very sharply, eventually peaking at 87% of the size of the C++ codebase. Following the sharp rise, it is accurate correct to say that Mozilla Firefox is written in JavaScript, with its Gecko layout engine making up the majority of the remaining C++ code. The only software built from C++ and distributed as “Mozilla Firefox” is the Gecko engine. The Mozilla Firefox that users experience exists as an effect of Gecko’s attendant XPFE components and the way they define Gecko’s interactions with the underlying operating system. XPFE’s JavaScript files are included as part of every Mozilla Firefox installation in a compressed file, but are

[22] This data was obtained from the pre-diff version of the Mozilla Corpus, which includes all files distributed by Netscape/Mozilla. Because topic modeling was only performed on the C++ files, officially the primary language of all three applications, the Javascript files were not part of the topic model.

otherwise unencrypted, allowing them to be extracted with a simple zip-compression utility and viewed by anyone. Presumably, anyone could directly modify or insert new features into the browser by editing the XML and JavaScript read by Gecko (see Figure 5.9). Both Mozilla Suite and Firefox, in this respect, are *configurations* of Gecko.

On a purely technical level, the Mozilla corpus demonstrates that a software's materiality is not stable; yet within the larger context of this study, the change introduced by XPFE highlights the disconnect between the interface and the computational present in personal computing software. Not only does the very structural definition of a piece of software subject to change over the course of its iterative development, but it can change in ways that leaves the interface largely intact. As Chapter 1 explores, one of the arguments for transparency proposed by computer scientists was the idea that a "soft" interface is only arbitrary linked to the algorithms it represents. Unlike "hard" interfaces, such as a steering wheel or a lever, there is a more direct link between the action of the user—his or her motion—and the mechanism it manipulates. Software, on the other hand, does not have this constraint and can be designed to resemble anything, producing a conceptual model that allows users to operate a computer system but which does not necessarily fully or accurately represent the computational operations taking place beneath it. In the case of the Mozilla Corpus, internal changes were only infrequently accompanied by external changes, allowing the conceptual model of a "web browser" to remain relatively intact.

III. Conclusion

Keeping up with the cultural work of a rapidly evolving software ecology is difficult, but this study shows that text analysis techniques, like topic modeling, can help us to pin down and trace events that may not be well documented in sources more familiar to archival

methodologies. Algorithmic tools are not perfect; however, acknowledging their epistemological limitations provides a space for researchers to shape their application in ways that respect the goals and concerns of humanistic inquiry. Topic modeling the entire version corpus without concern for duplication would have produced topic vectors but with trends that would have been more blunted. Given Thomas' findings regarding the accuracy of a diff-wrapped LDA, compared to standard pre-preprocessing, it is possible that several of the trends or surprising moments discussed above may not have been visible. Facing this constraint of LDA opened a space for reflection on digital media scholarship which in turn led to methodological solutions that accounted for the unique material qualities of software's production. The resulting data also serves as an argument for the need for more historical studies of software's development, whether through this methodology or others. Comparative media analysis served as an excellent technique for applying critical theories developed for literary study to television, film, and digital media. Here, too, lessons from print culture and the computational study of print corpora helped to articulate the problem of a sprawling assemblage into a bibliographic problem. As the digital humanities moves forward, dialogue between those working on computational methodologies in literary analysis and software studies could lead to a sharing of techniques, revealing new ways of approaching critical problems in the two fields.

Mozilla Firefox's incorporation of HTML, JavaScript, and XML into its codebase shows just how much change is possible to go unnoticed beneath transparent software interfaces. The Internet is no longer just something used to "surf," download media, or play games; it is fast becoming a required aspect of personal computing, and software is being reshaped to incorporate not just its network protocols but also its content model. Closed source software developers are also been pushing to integrate web technologies into their product. Microsoft, for example, offers

an “Office 365” program that presents it applications as subscription services, rather than objects for which users purchase licenses. Even the latest Office includes built-in cloud storage, inviting users to store all their data on Microsoft’s servers. Developments like these are moving personal computers away from the idea of appliances that reside in our homes and more towards something resembling a semi-permanent conduit between our lives and the economic interests that can almost instantaneously—our Internet infrastructure perhaps isn’t quite that fast yet—control our relationship to them with the press of a button. Web languages are, in turn, being reshaped to resemble more robust application development languages, allowing for a style of programming that mixes languages to effect something like the structural organization of C++ and Java but without the need for developers to worry about system management problems like exception handling, build dependencies, or compiler errors. As we enter the era of cloud computing, Internet technologies are expected to replace personal computers, returning hardware to something resembling the mainframe-centric architecture that prevailed prior to the 1980s.

Within the broader context of this project, this chapter shows the extent to which transparent software can undergo significant shifts in structure that are not visible through a perspective shaped by user interfaces. Unless we take steps to draw back the veil of commercial models of user-friendliness soon, it may be too late for anyone other than developers to have much of a say in what sort of automatic systems are built into the next generation of personal computer technology. Even if copyright and other intellectual property protections prevent major commercial software from being open to the historical analysis performed here, there is still good reason to believe that studying related free and open source technologies can help us to understand better the software ecologies they share with closed software. It’s important that the cultural dimension of these technologies be addressed openly and frankly, the same way that new

features or the latest technical specifications are now. Histories of software must do more than recount the technical achievements or corporate strategies of developers. As this chapter has shown, it is possible to leverage computational power as a meta-critical tool. The problems of scope and scale posed by software history can be accounted for using digital tools in methodologies informed by critical and interpretive theory.

Appendix A – Tables and Figures for Chapter 4

Table 4.1: Information regarding the xfce4 dependency corpus

Dependency tree for Xfce, a lightweight GTK+ desktop environment	
Gathered:	Dec. 2012 from Debian's http.us mirror
Version:	4.6.2, with Debian 6.0 patches
Languages:	C/C++
Packages:	223 (174 containing code in target languages)
Files containing code:	16,017
Lines of code:	255,571
Total tokens:	16,538,166
Less stopwords:	6,460,629

Table 4.2: Sample topics from each of the 6 codings

User applications		GUI and graphics libraries		Multimedia libraries	
Topic 0	Topic 17	Topic 1	Topic 7	Topic 4	Topic 10
Gtk	thunar	dpi	xfce	mixer	psf
Ical	icon	req	icon	snd	channel
button	view	xkb	menu	pcm	pcm
clock	model	xcm	panel	track	frame
appt	gtk	color	plugin	seq	seek
alarm	dialog	font	dialog	alsa	marker
widget	rename	rep	widget	port	bufferlen
Box	volum	cony	channel	card	sampl
Data Encoding		System Management		Command line utilities	
Topic 3	Topic 22	Topic 9	Topic 14	Topic 31	
xml	node	pci	udev	ncur	
parser	gam	conn	queue	parm	
tok	idx	auth	enum	menu	
glade	reg	xcb	monitor	screen	
enc	dfa	cap	subsystem	export	
entity	tree	repli	lang	trace	
pool	acl	protocol	serial	color	
decl	subscript	host	scsi	attr	

Table 4.3: Sample Pearson results for direct dependencies. First row contains the higher level package and dependencies appear on subsequent rows. The number at the beginning of the package name indicates the layer assigned by Pajek's depth calculation.

2-xfwm4		3-hal	
4-libwnck22	0.9973	4-libblkid1	0.0214
4-libxfcegui4-4	0.8405	4-pciutils	0.0134
5-libglade2-0	0.1557	4-udev	0.0488
5-libstartup-notification0	0.8790	4-usbutils	0.9966
5-libxfconf-0-2	0.0201	5-libusb-0.1-4	0.9977
7-libdbus-glib-1-2	0.0343	6-libhal-storage1	0.9994
7-libgtk2.0-0	0.2180	7-libdbus-glib-1-2	0.0171
7-libxfce4util4	0.0328	7-libhal1	0.9353
8-libpango1.0-0	0.0270	8-dbus	0.0120
8-libxcomposite1	0.0255	9-libglib2.0-0	0.0200
8-libxdamage1	0.0130	9-mount	0.0213
8-libxrandr2	0.0303	10-libdbus-1-3	0.0123
9-libglib2.0-0	0.0290	12-libexpat1	0.0209
9-libxext6	0.0239		
10-libxrender1	0.0341		
11-libx11-6	0.0235		

Table 4.4: Sample Pearson results for intra-layer neighbors. The first row is the package to which packages on all subsequent rows were compared. The number at the beginning of the package name indicates the layer assigned by Pajek's depth calculation.

2-xfdesktop4		7-dbus-x11	
2-gtk2-engines-xfce	0.4120	7-libdbus-glib-1-2	0.9172
2-orage	0.0274	7-libgmp3c2	0.0271
2-thunar-volman	0.4377	7-libgtk2.0-0	0.0434
2-xfce4-appfinder	0.9657	7-libhal1	0.3149
2-xfce4-mixer	0.7119	7-libudev0	0.0310
2-xfce4-session	0.6102	7-libvorbis0a	0.0262
2-xfce4-utils	0.8335	7-libxfce4util4	0.0084
2-xfwm4	0.1291	7-libxmu6	0.0302
		7-libxpm4	0.0295
		7-util-linux	0.0586

Figure 4.1: An example layer model, taken from the MeeGo mobile GNU/Linux distribution developer documentation.
See: https://meego.com/developers/meego-architecture/meego-architecture-layer-view

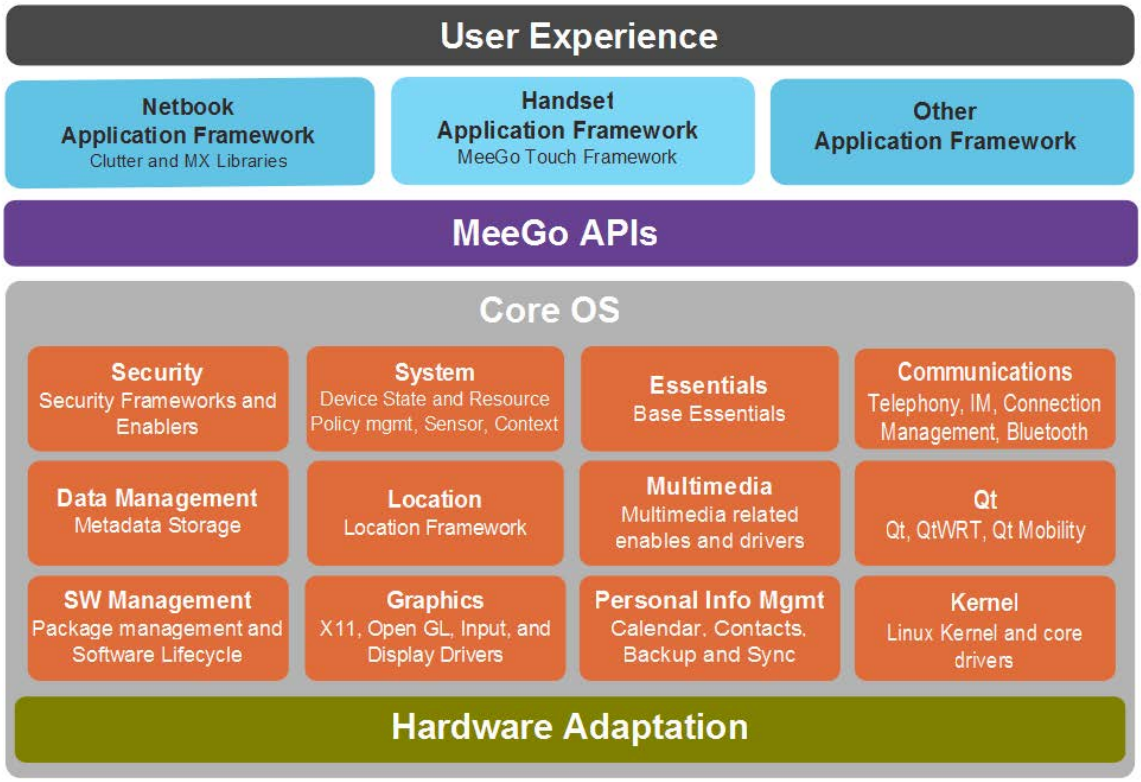


Figure 4.2: A layer diagram illustrating the user's relationship to the interface and the rest of the system according to transparency. This diagram fits well with Manovich's culture/computation model.

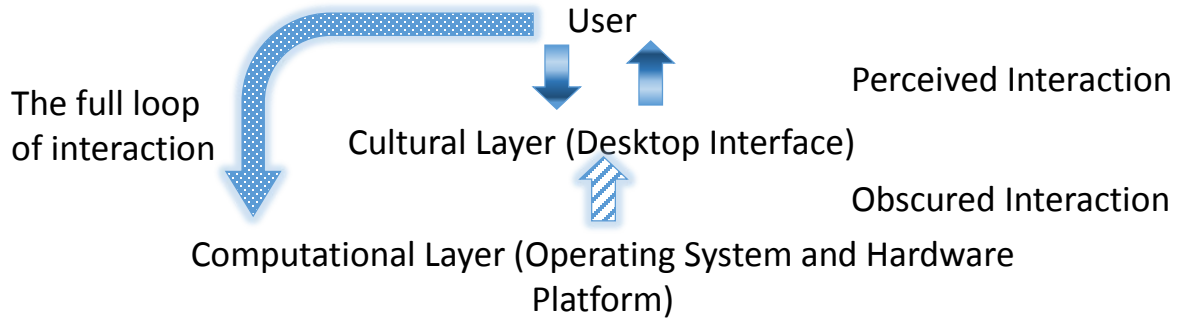


Figure 4.3: Dependency tree for the xfce4 package in Debian 6.0. This tree was calculated using Pajek's acyclic hierarchy depth partition algorithm. Layers represent maximum distance from the xfce4 package (placing all packages on the layer directly below the lowest package that depends upon them). Partition colors only represent layer groupings.
[Full resolution available at: http://mblack.us/chap4/]

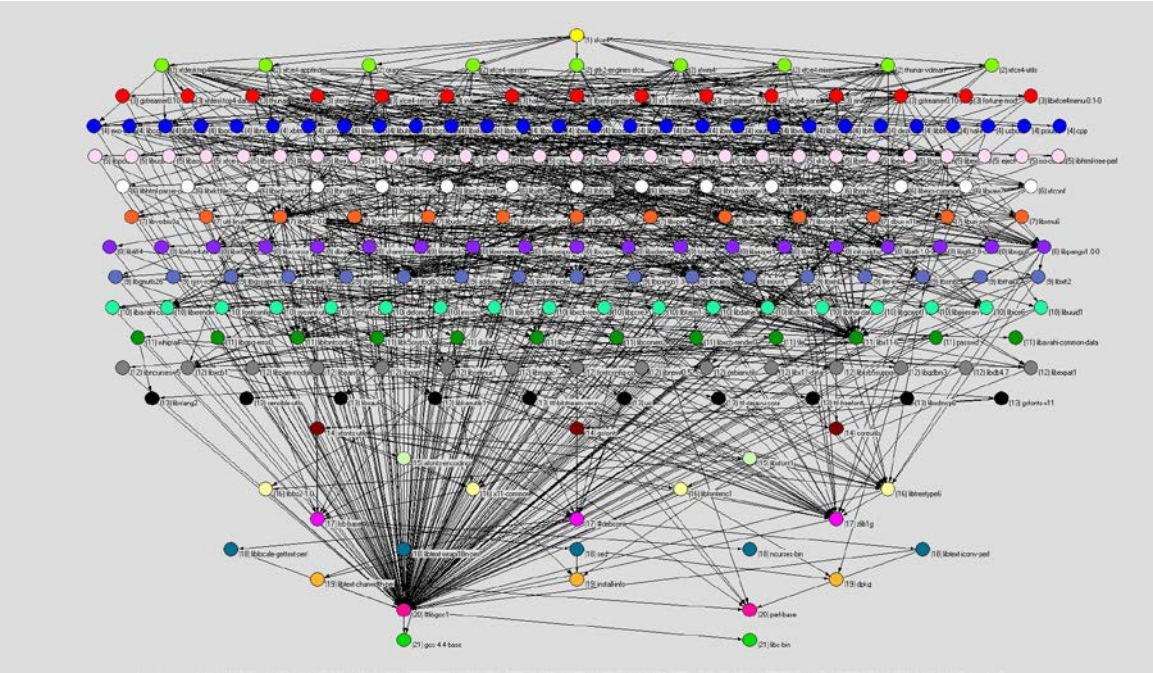


Figure 4.4: Trimmed Tree for xfce4 package in Debian 6.0. This tree uses the same layering data from Figure 3, but all packages not containing C/C++ code are removed. Packages are coded according to role represented by topic with the highest number of tokens and x-axis position is optimized to show clustering. [Full resolution available at: http://mblack.us/chap4/]

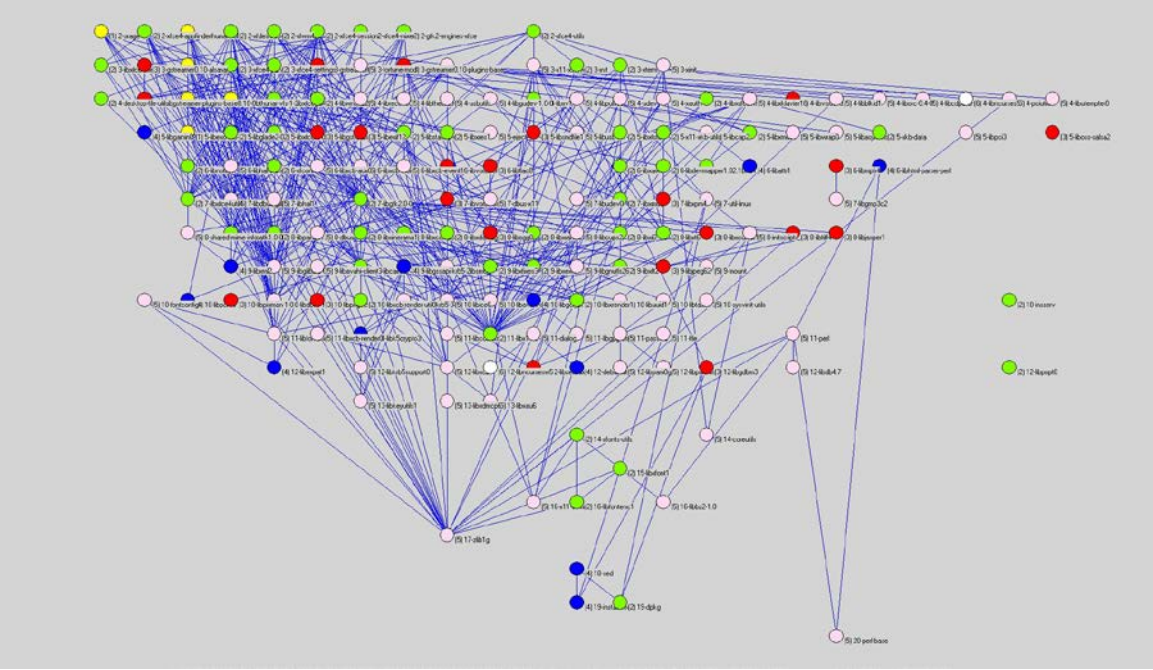
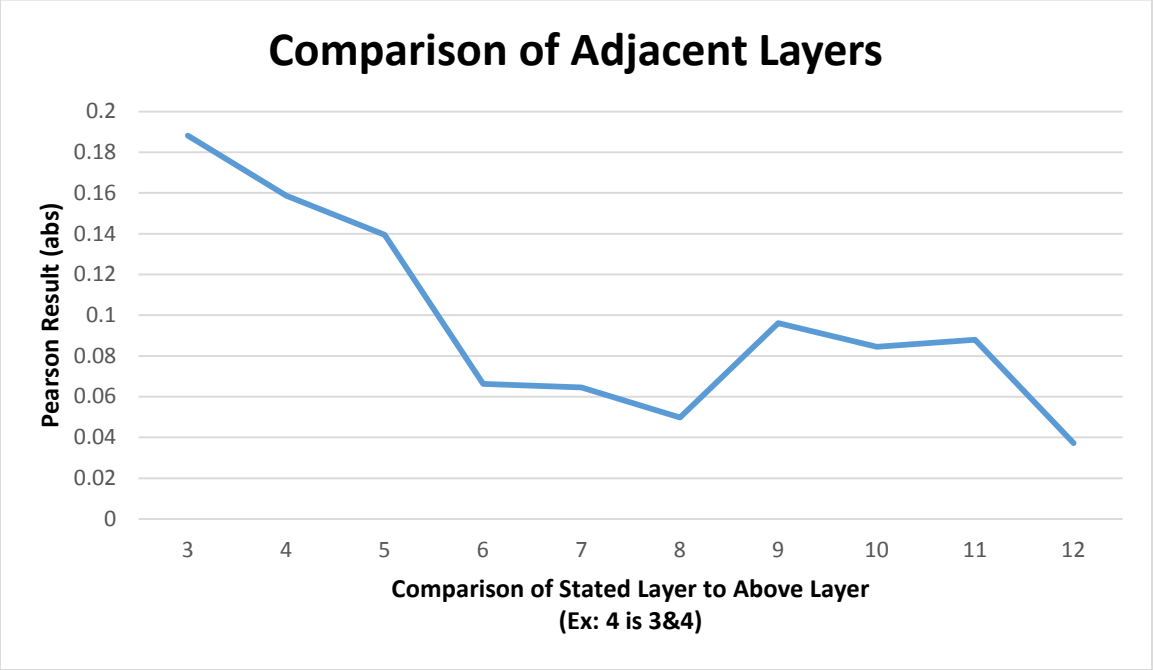


Figure 4.5: Layer to layer comparisons. The first graph shows the Pearson results when each layer is compared to the one immediately above it.



The second graph shows the same results superimposed on top of package counts for each layer.

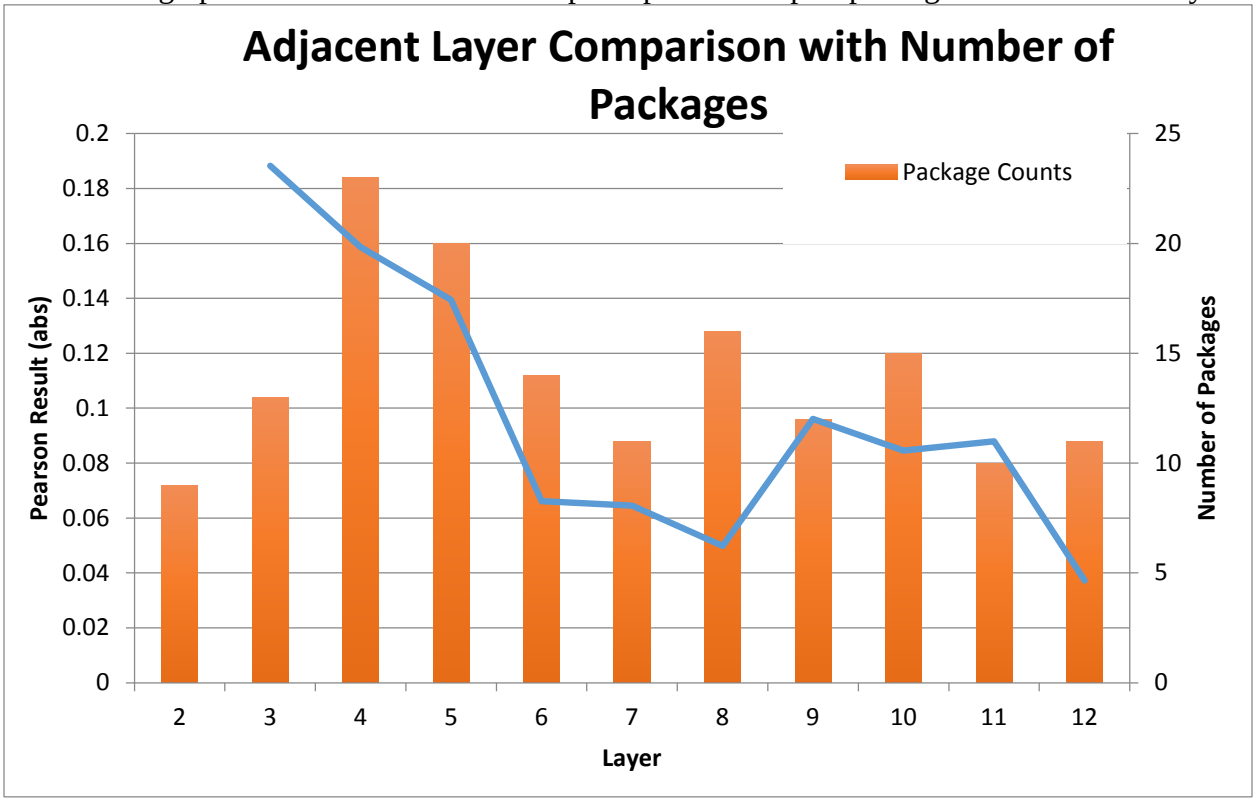
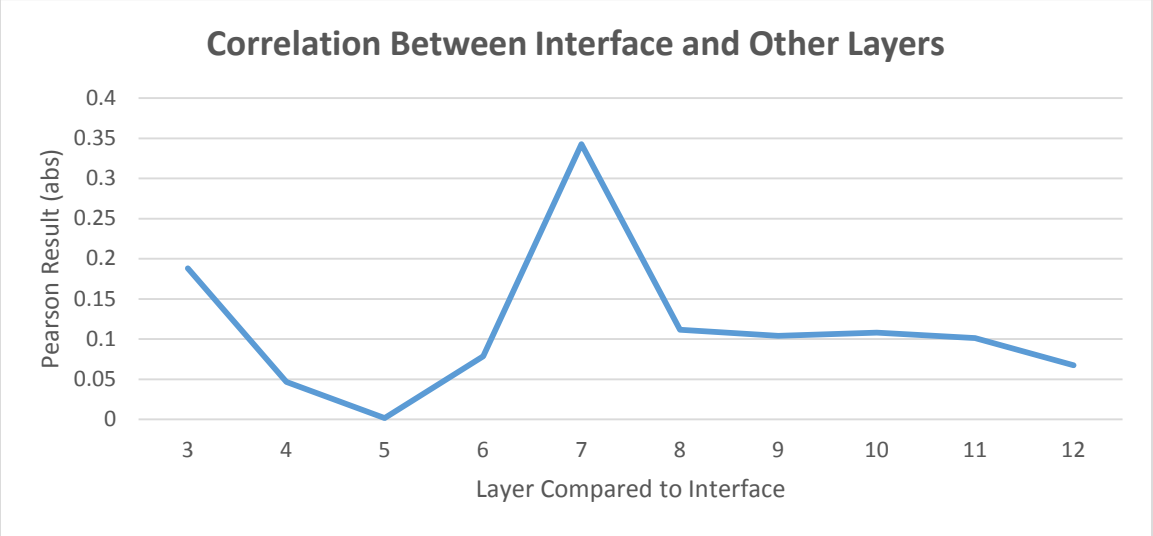


Figure 4.6: Interface to layer comparisons. The summed vector for all packages on layer 2 was compared to the summed vectors for all other layers. The first graph shows Pearson results for all packages on layers 2-12. The strong spike at layer 7 is produced by the libgtk2, a graphic library that every GUI application depends upon.



The second graph shows Pearson results when libgtk2 is removed from the layer 7 vector.

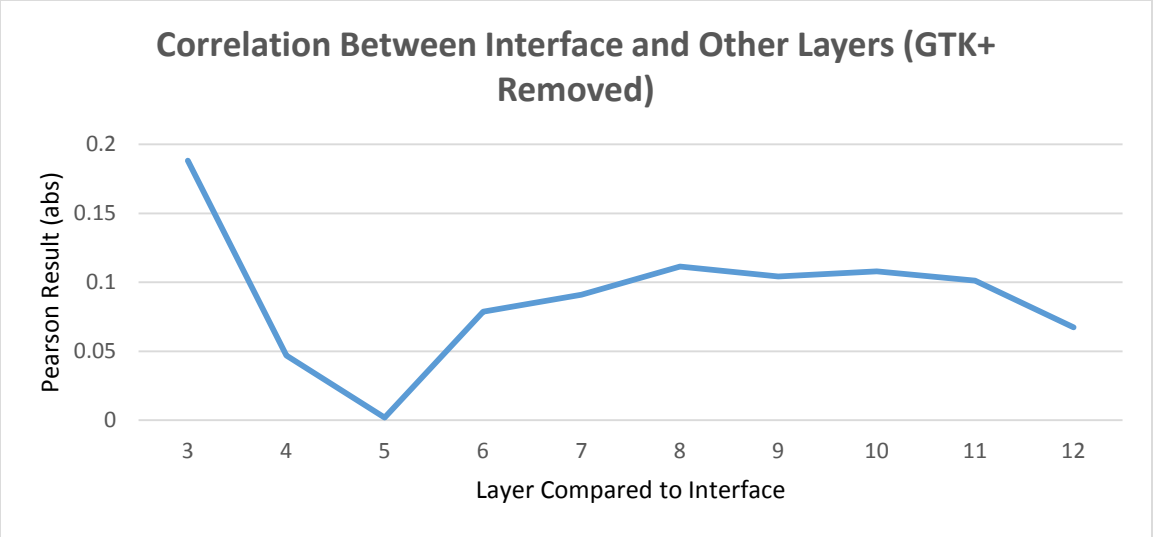


Figure 4.7: Similarity Tree produced using global Pearson results. Each edge in the network represents a result of 0.3 or higher. Only new relationships are rendered; direct dependencies are not rendered regardless of Pearson result. Packages coded according to role represented by topic with the highest number of tokens.
[Full resolution available at http://mblack.us/chap4/]

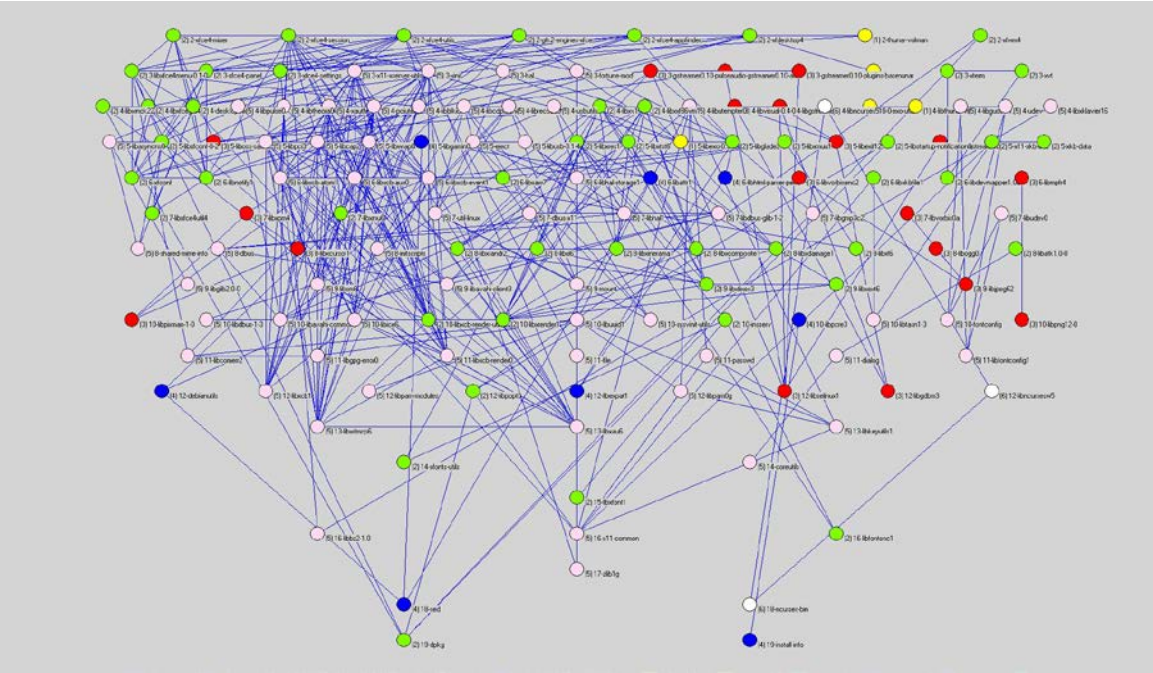


Figure 4.8: Layer to layer package relationship counts for the Similarity Tree.

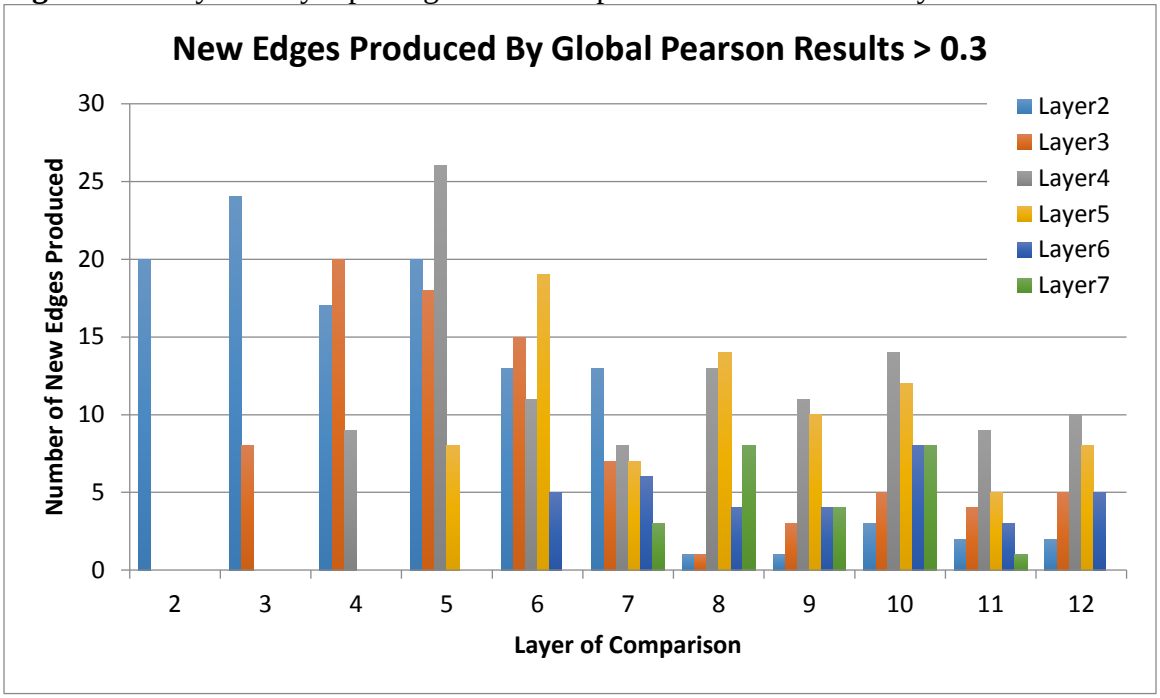
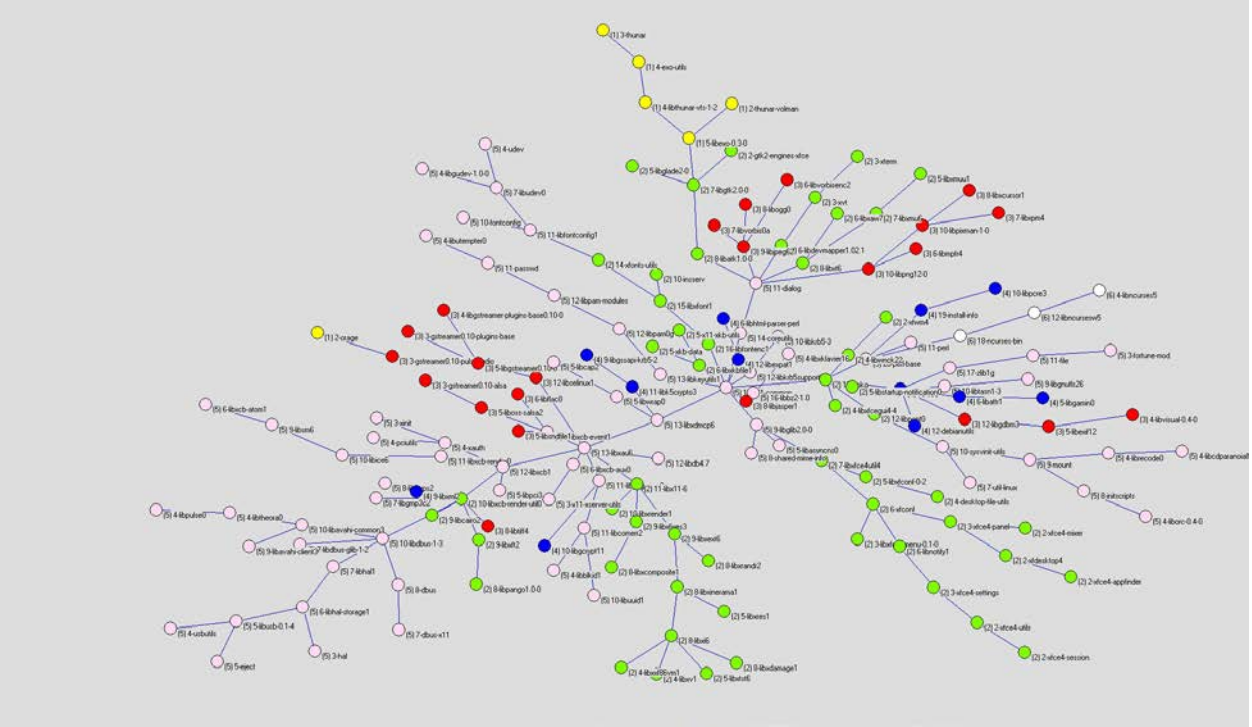


Figure 4.9: Map produced via top Pearson pairings. Full resolution available at http://mblack.us/chap4/



Appendix B – Tables and Figures for Chapter 5

Table 5.1: General information about the Mozilla Corpus

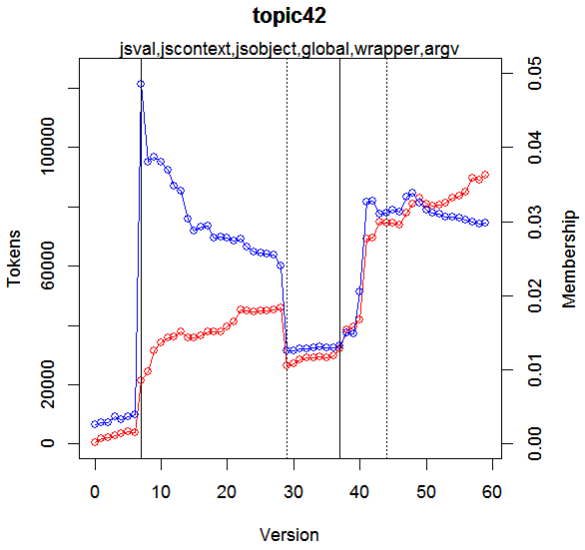
Version Label	Version Number	Release Date	Size (Full)	Net Change (Full)	Net Change (C++)	# of Tokens
Mozilla Communicator						
00	None	03/31/98	40203077	N/A	N/A	233285
01	None	04/08/98	59131622	18928545	9694228	692924
02	None	04/29/98	65089258	5957636	548493	714233
03	None	06/03/98	70166069	5076811	1223271	753252
04	None	07/28/98	84099725	13933656	8086869	1132744
05	None	09/04/98	90469752	6370027	330190	1144931
06	None	10/08/98	83588207	-6881545	-1708705	1048406
Mozilla Application Suite						
07	(M1?)	12/11/98	44647824	N/A	N/A	674816
08	(M2?)	01/28/99	55504133	10856309	4673001	865565
09	M3	03/19/99	65940062	10435929	4740616	959360
10	M4	04/15/99	70438485	4498423	2334716	1031601
11	M5	05/05/99	73321774	2883289	1766415	1113046
12	M6	05/29/99	85172946	11851172	2240285	1190649
13	M7	06/22/99	86226782	1053836	1453313	1269994
14	M8	07/16/99	90528214	4301432	2082572	1347142
15	M9	08/26/99	94343632	3815418	1752674	1342563
16	M10	10/08/99	99242381	4898749	413658	1393442
17	M11	11/16/99	105517092	6274711	1403584	1477091
18	M12	12/21/99	110274354	4757262	1731966	1464024
19	M13	01/26/00	108091563	-2182791	-216838	1540253
20	M14	03/01/00	112325279	4233716	1751420	1630439
21	M15	04/18/00	117337663	5012384	1845194	1783563
22	M16	06/13/00	123233166	5895503	3866597	1839983
23	M17	08/20/00	141779386	18546220	1289320	1882548
24	M18	10/12/00	145153396	3374010	903169	1896079
25	0.6	12/06/00	135022311	-10131085	271163	1917088
26	0.7	01/09/01	152227947	17205636	507036	1936538
27	0.8	02/14/01	152513233	285286	293603	2090149
28	0.9	05/07/01	170953881	18440648	3763559	2274789
29	1.0	06/05/02	224325235	53371354	6189989	2332033
30	1.1	08/26/02	228137982	3812747	1363139	2402289
31	1.2	11/26/02	228896289	758307	1737235	2433636
32	1.3	03/13/03	229003326	107037	725868	2436192

Table 5.1 (cont.)

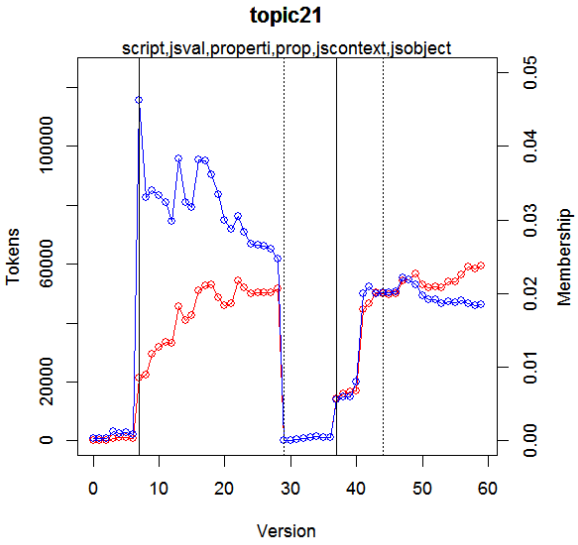
33	1.4	06/30/03	230873828	1870502	157801	2427898
34	1.5	10/15/03	184967631	-45906197	-68683	2447614
35	1.6	01/15/04	188991165	4023534	290254	2503244
36	1.7	06/17/04	206097164	17105999	1299795	2545786
Mozilla Firefox						
37	1.0	11/09/04	197018982	N/A	N/A	2809989
38	1.5	11/30/05	208257684	11238702	5285960	2155079
39	2.0	10/24/06	218235493	9977809	2380870	2237129
40	3.0	06/17/08	227187445	8951952	-15144818	2241148
41	3.5	06/30/09	278296889	51109444	2564959	2531559
42	3.6	01/21/10	280683836	2386947	690404	2504039
43	4.0	03/22/11	346351093	65667257	5805397	2483528
44	5.0*	06/21/11	352386092	6034999	-729812	2469320
45	6.0	08/16/11	347554884	-4831208	-516552	2456907
46	7.0	09/27/11	349362417	1807533	-769400	2511091
47	8.0	11/08/11	354846376	5483959	-253822	2686495
48	9.0	12/20/11	364515482	9669106	1535342	2694930
49	10.0	01/31/12	372937453	8421971	4338895	2714671
50	11.0	03/13/12	385911428	12973975	-55856	2740373
51	12.0	04/24/12	388876784	2965356	472535	2790036
52	13.0	06/05/12	393197798	4321014	576292	2864587
53	14.0.1	07/17/12	407165259	13967461	1061975	2891327
54	15.0	08/24/12	395983732	-11181527	-2924526	2976568
55	16.0	10/09/12	429430729	33446997	552219	3167788
56	17.0	11/20/12	435263141	5832412	1455213	3182711
57	18.0	01/08/13	465517793	30254652	4674165	3216161
58	19.0	02/19/13	471549259	6031466	292543	233285
59	20.0	04/02/13	476262707	4713448	774360	692924

*** Mozilla begins its rolling release, “every six weeks,” model.**

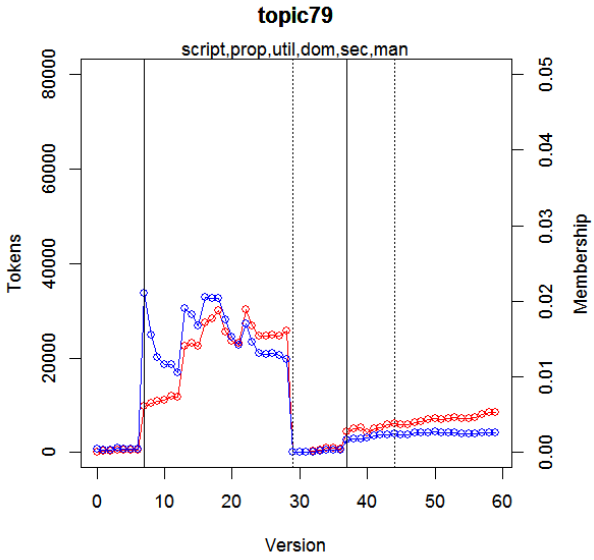
Figure 5.1: A comparison of the XPConnect topic. XPConnect serves as the internal interface between the Mozilla's Javascript interpreter and its Cross Platform Component Object Model (XPCOM), which serves an abstract system management module (interfacing with memory management, file system, and internal messaging libraries).



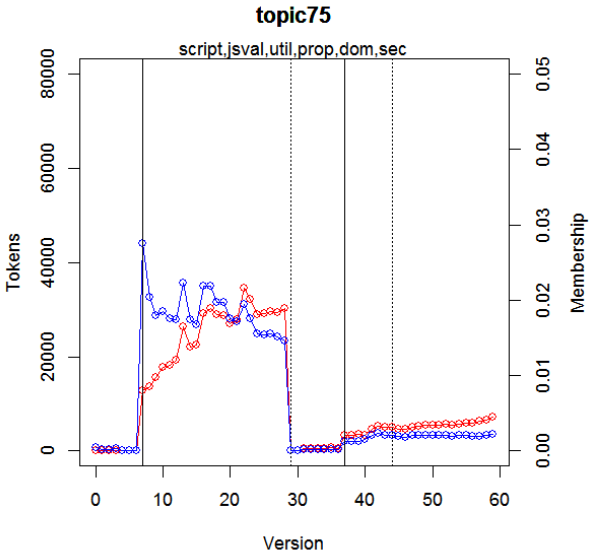
Model size: 50 topics



Model size: 60 topics

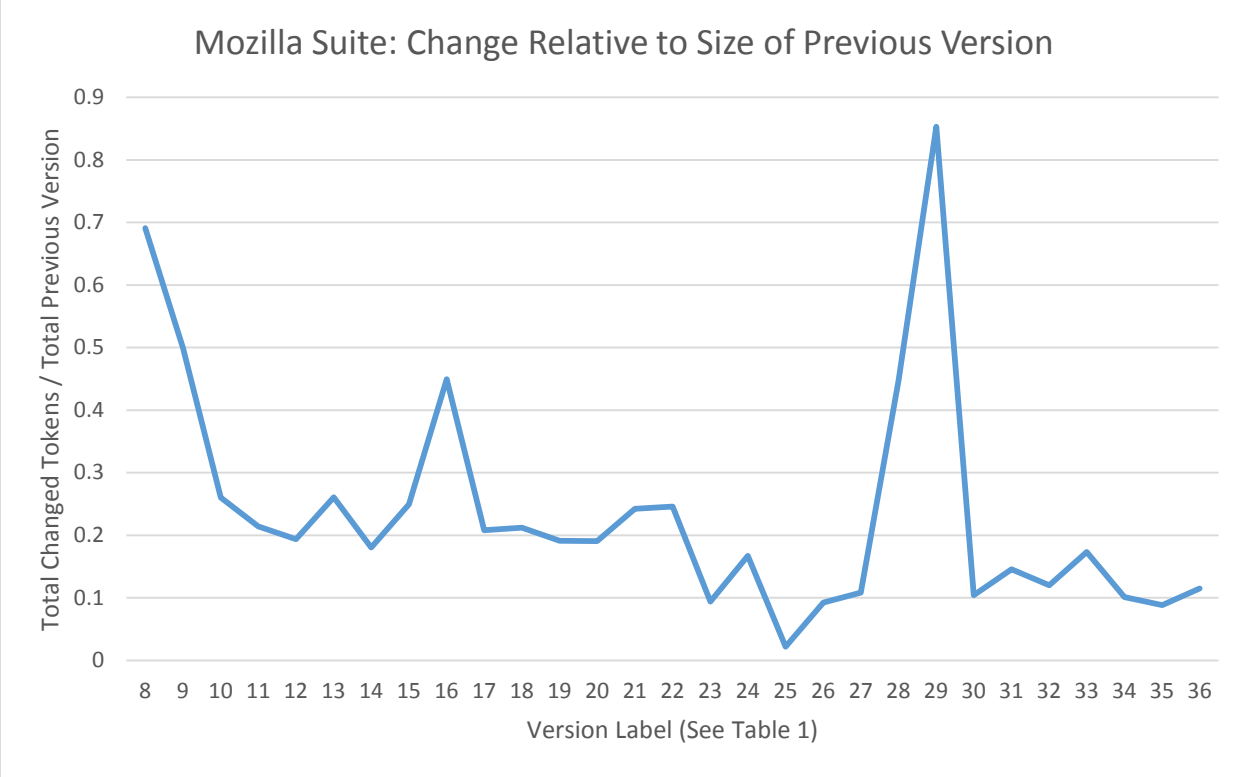


Model size: 80 topics

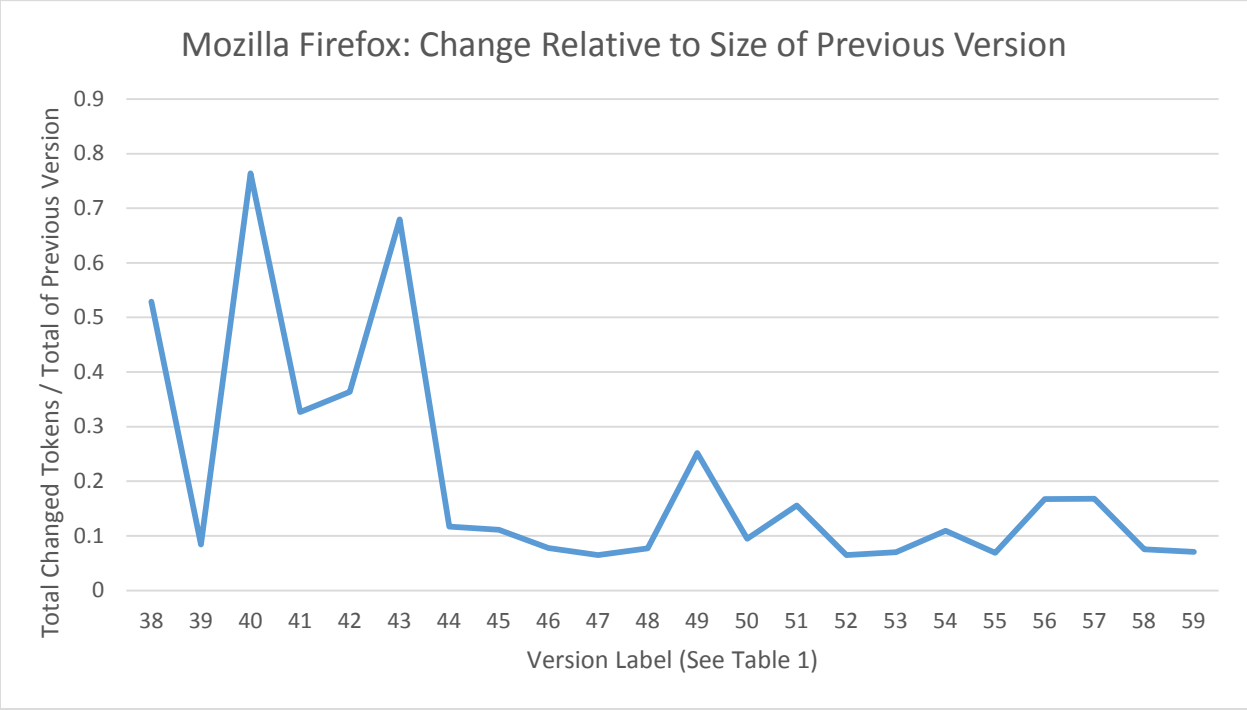


Model size: 100 topics

Figure 5.2: Graphs for Mozilla Suite and Mozilla Firefox of the total change in code base over time, calculated as the sum of the tokens changed (ABS(δ-remove) + δ-add) relative to the total number of tokens in the previous version.



The average change between versions during the Mozilla Suite era is 23.86%.



The average change between versions during the Mozilla Firefox era is 20.41%.

Figure 5.3: Topic vectors related to e-mail show a sharp decline at label 40. This decline coincides both with a significant change in the size of the C++ code base and with an announcement by Mitchell Baker, head chairperson for the Mozilla Foundation, that new resources would be devoted to Thunderbird, Firefox's companion e-mail client. Because both use the Gecko layout engine, these graphs suggest that the change in direction was as much about streamlining Firefox as it was about further support for Thunderbird. In other words, these graphs show the removal of application-specific code from Gecko. This change is discussed in detail later in this chapter.

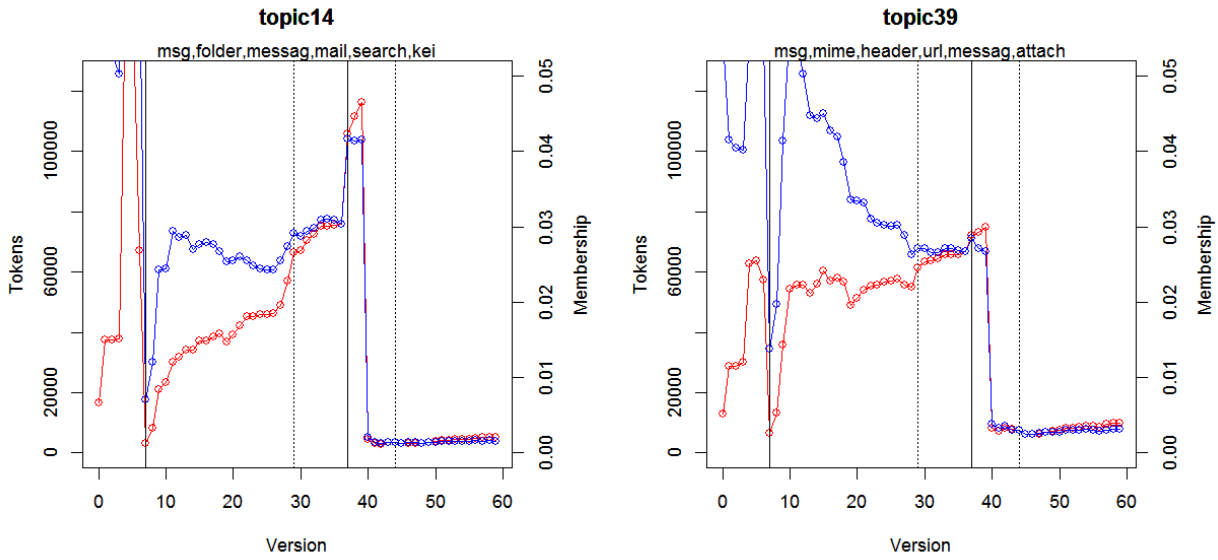


Figure 5.4: Three topics representing GUI rendering that exhibit a downward trend. While topics 22 and 27 only show this trend after the release of Firefox, topic 56 highlights Netscape's use of different user-interface implementations for each of its supported platforms.

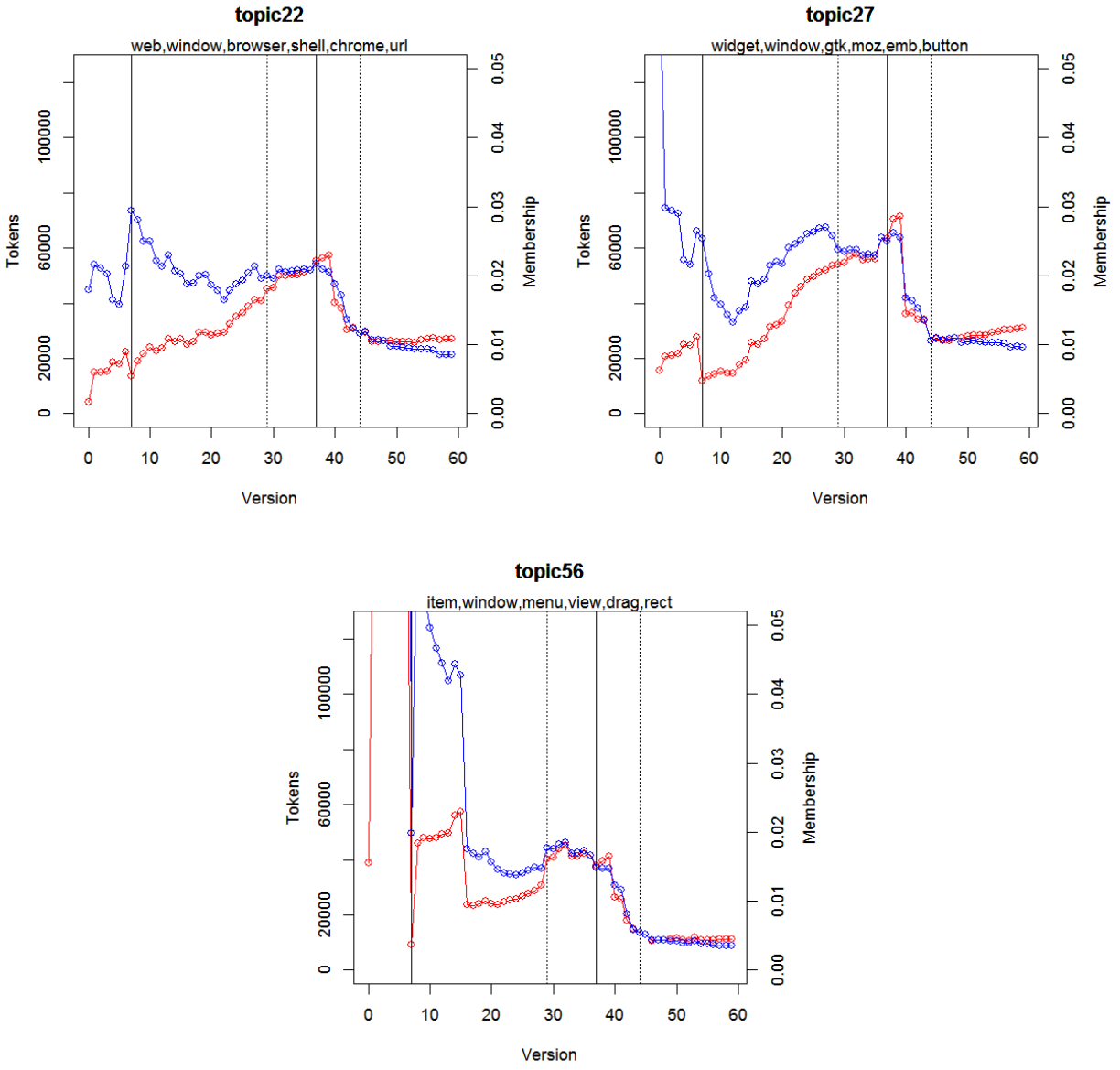


Figure 5.5: Examples of graphics components with steady or sharp upward trends.

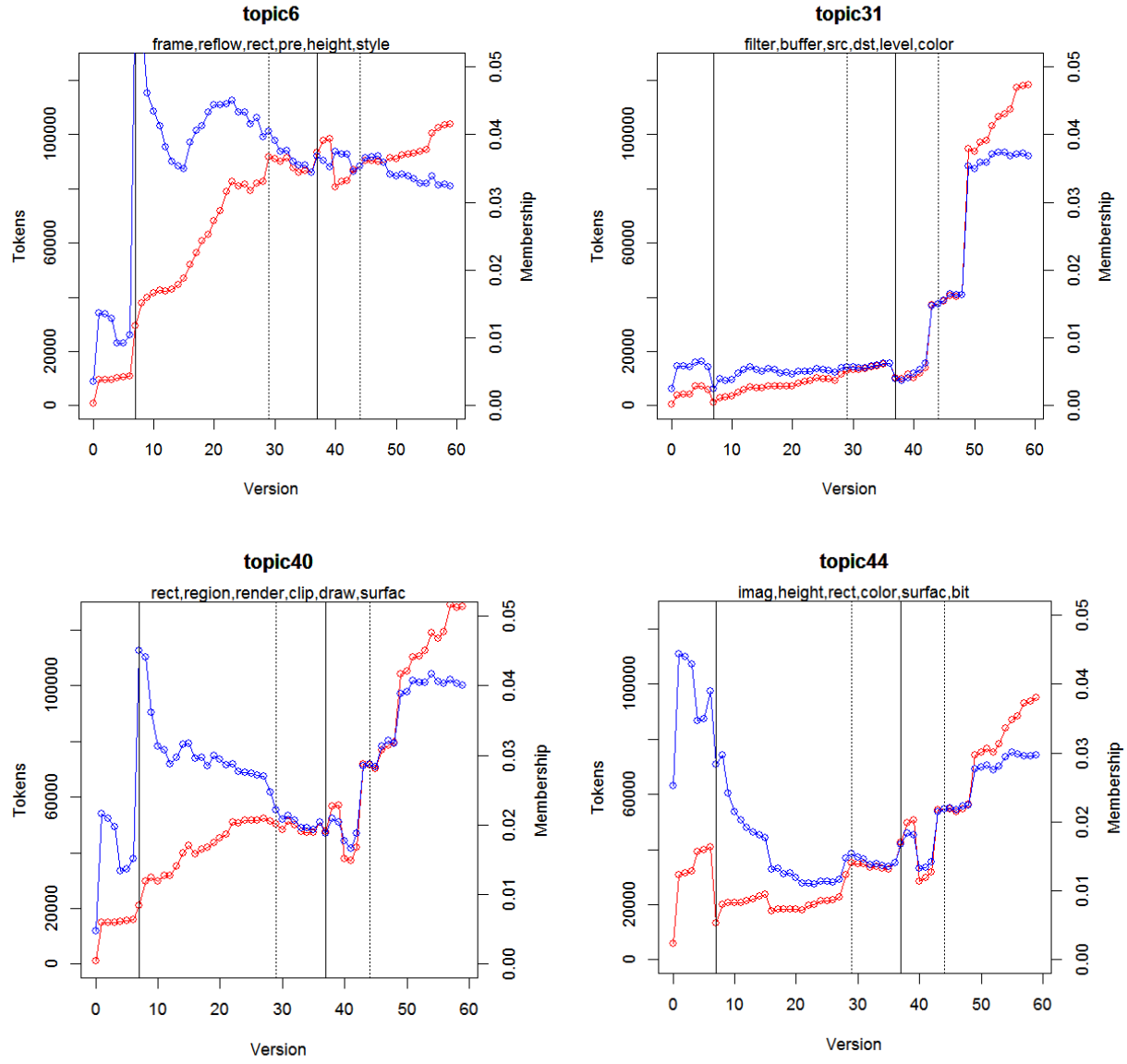
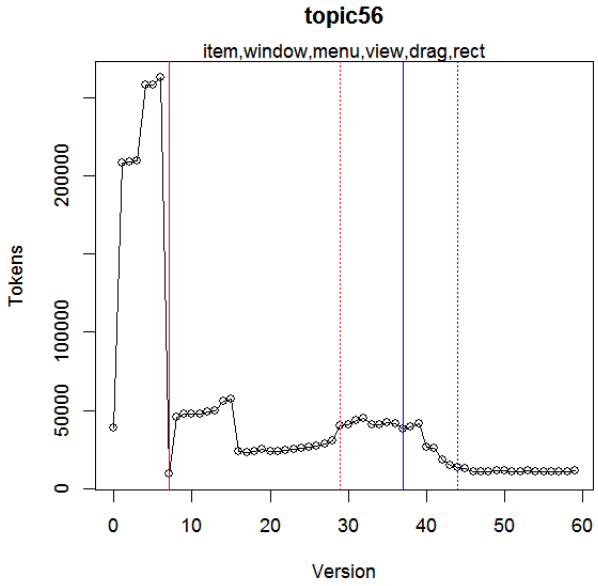
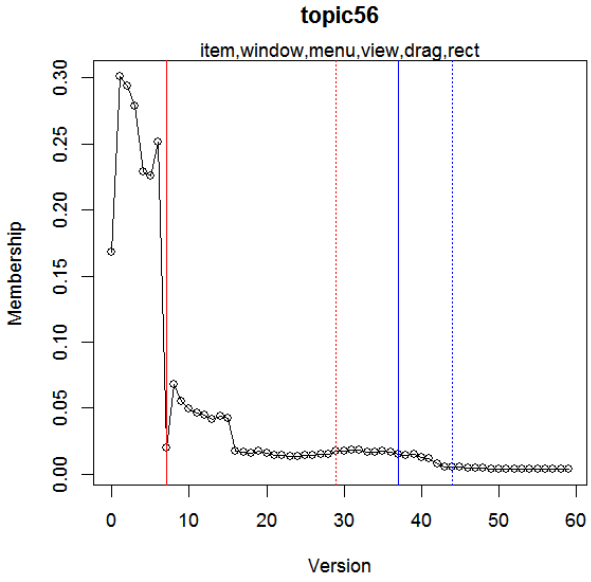


Figure 5.6: Unscaled versions of topic 56's vector that show its peak both as token count (left) and as percentage of C++ codebase (right).

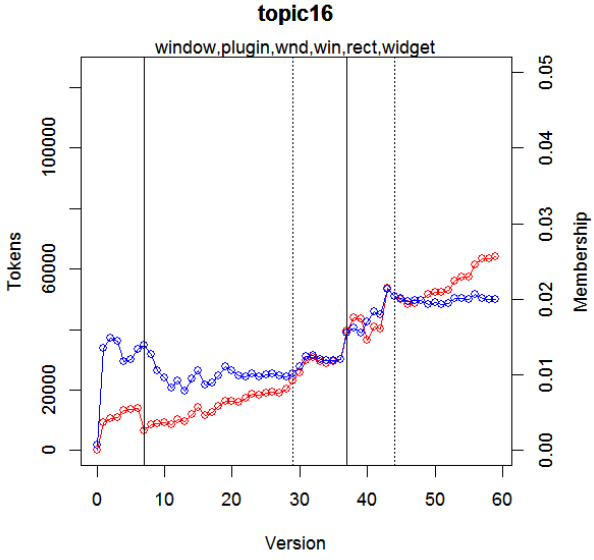


Unscaled Token Count

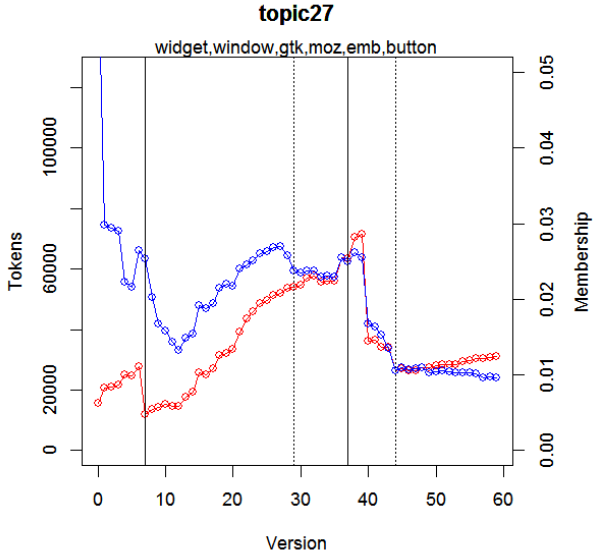


Unscaled Membership Percentage

Figure 5.7: Topics representing the lower layers of XPFE that interact with OS-specific graphics libraries. The discrepancy in trend during the Firefox era likely represents the different amounts of code necessary to manage windows on the two platforms.



Topic 16's terms: window, plugin, wnd, win, rect, widget, instanc, mou, param, key, messag, msg, point, dispatch, proc, handl, timer, hwnd, cursor, height, child, scroll, bound, statu, screen



Topic 27's terms: widget, window, gtk, moz, emb, button, browser, menu, mou, focu, label, kei, dialog, signal, statu, callback, drag. cursor, shell, ctx, debug, app, titl, dlg, mask

Figure 5.8: Size of the code-base of each of the major languages used in the Mozilla Corpus. XML includes all known variants used by Netscape/Mozilla (XML, XUL, RDF)

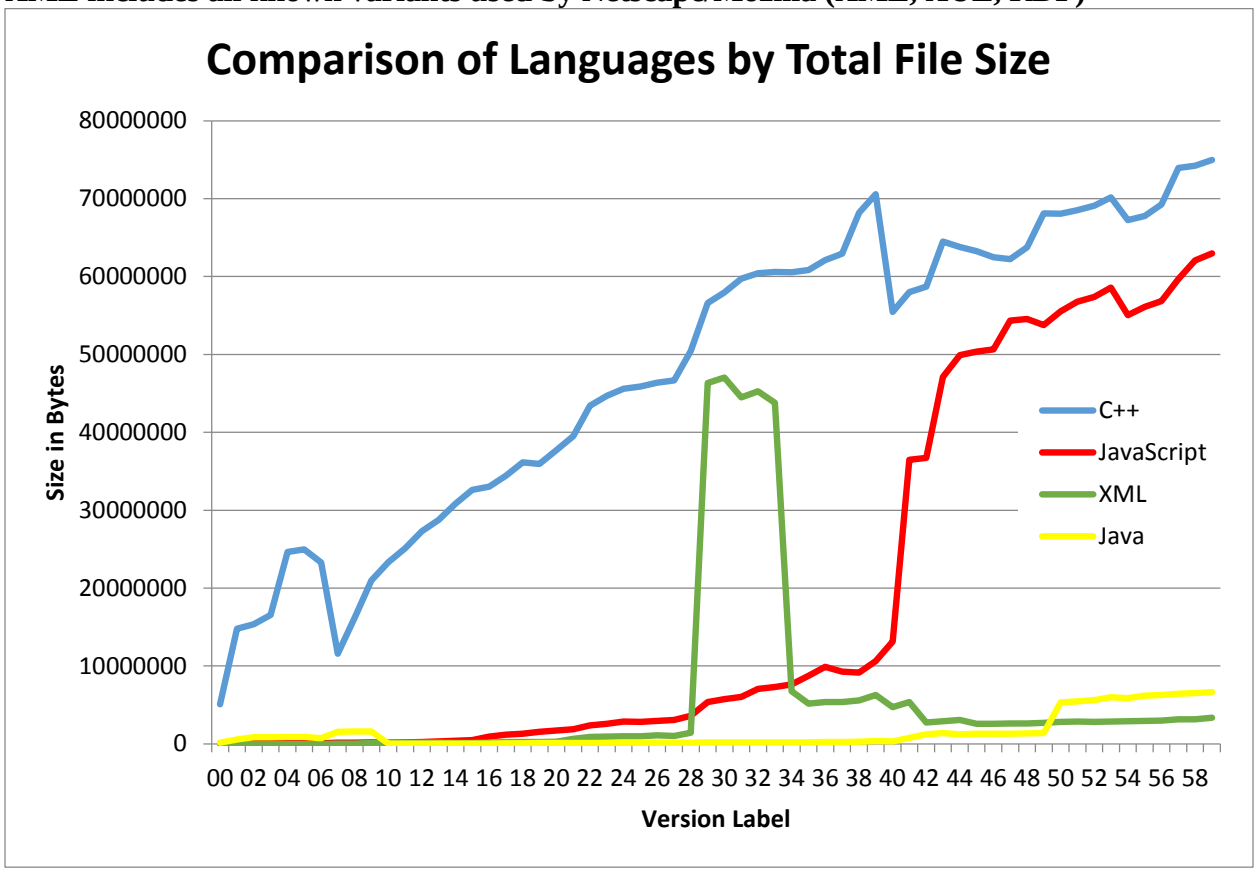
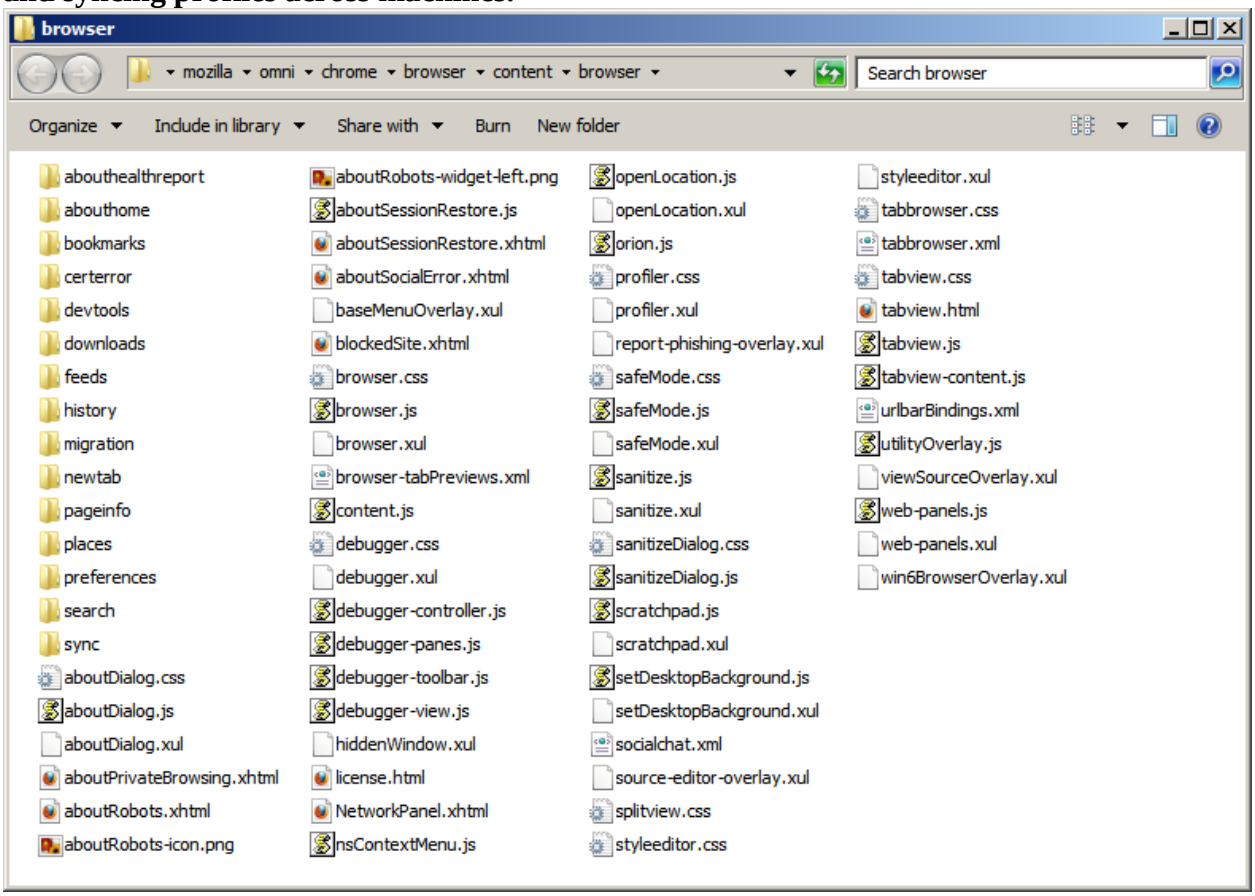


Figure 5.9: Some the Firefox browser components extracted from omni.ja. In addition to the basic browser window, these scripts also control common browsing utilities tabbed browsing, downloads, RSS feeds, bookmarks, browser-wide preferences menus, history, and syncing profiles across machines.



References

Anderson, John J. “Apple Macintosh: Cutting Through the Ballyhoo.” *Creative Computing* (July 1984). Web. 17 Sept 2011.

Anderson, Nate. “How an omniscient Internet ‘sextortionist’ ruined the lives of teen girls.” *Ars Technica*. Web. 16 July 2012.

Apple Computer. *Macintosh Human Interface Guidelines*. Reading: Addison-Wesley, 1992. Print.

Atwood, Margaret. *The Handmaid’s Tale*. Boston: Houghton Mifflin, 1986. Print.

Balsamo, Anne. *Technologies of the Gendered Body: Reading Cyborg Woman*. Durham: Duke UP, 1996. Print.

Barad, Karen. *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*. Durham: Duke UP, 2007. Print.\

Barnet, Belinda. “Machine Enhanced (Re)minding: the Development of Storyspace.” *Digital Humanities Quarterly* 6.2 (2012). Web. 8 Oct 2012.

Bergin, Thomas J. and Richard G. Gibson. Eds. *History of Programming Languages*, Vol 2. New York: ACM Press, 1996. Print.

Black, Michael. , “Narrative and Spatial Form in Digital Media: A Platform Study of the SCUMM Engine and Ron Gilbert's *The Secret of Monkey Island* .” *Games and Culture* 7.3 (May 2012): 209-237. *Sage*. Web. 3 Feb 2014.

Blei, David M., Andrew Y Ng, and Michael I Jordan. “Latent Dirichlet allocation .” *Journal of Machine Learning* 3 (2003): 993-1022. Web. 7 July 2013.

Bogost, Ian. *Unit Operations: An Approach to Videogame Criticism*. Cambridge: MIT Press, 2006. Print.

Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Redwood City: Benjamin/Cummings, 1994. Print.

Bolter, J. David. *Writing Space: The Computer, Hypertext, and the Remediation of Print.* 2nd Ed. Mahwah: Lawrence Erlbaum Associates, 2001. Print.

Bolter, J. David and Diane Gromala. *Windows and Mirrors: Interaction Design, Digital Art, and the Myth of Transparency*. Cambridge: MIT Press, 2003. Print.

Bolter, J. David and Richard Gruin. *Remediation: Understanding New Media*. Cambridge: MIT Press, 2002. Print.

Brand, Stewart. "Spacewar: Fanatic Life and Symbolic Death Among the Computer Bums." *Rolling Stone*. Web. 30 June 2011.

Bredehoft, Thomas A. "The Gibson Continuum: Cyberspace and Gibson's Mervyn Kihn Stores ." *Science Fiction Studies* 22.2 (1995): 252-263. *JSTOR*. Web. 23 June 2011.

"Bruce Sterling: Just a Sci-Fi Guy." *Locus* (May 1988): 6+. Print.

Bukatman, Scott. *Terminal Identity: The Virtual Subject in Postmodern Science Fiction*. Durham: Duke UP, 1993. Print.

Card, Stuart K., Thomas Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Hillsdale: Lawrence Erlbaum Associates, 1983. Print.

Campbell-Kelly, Martin and William Aspray. *Computer: A History of the Information Machine*. New York: Basic Books, 1996. Print.

Carrol, John M. "The Adventure of Getting to Know A Computer." *Computer* 15.1 (1982): 49-58. *IEEEXplore*. Web. 25 June 2012.

Ceruzzi, Paul E. *A History of Modern Computing.* 2nd Ed. Cambridge: MIT Press, 2003. Print.

Chen, Tse-Hsun, et al. "Explaining Software Defects Using Topic Models." *2012 9th IEEE*

Working Conference on Mining Software Repositories (Proceedings). Web. 10 Aug 2012.

Chun, Wendy Hui Kyong. *Programmed Visions: Software and Memory*. Cambridge: MIT Press, 2011. Print.

--. *Control and Freedom: Power and Paranoia in the Age of Fiber Optics*. Cambridge: MIT Press, 2006. Print.

Csicsery-Ronay, Jr., Istvan. "Antimancer: Cybernetics and Art in Gibson's 'Count Zero,'" *Science Fiction Studies* 22.1 (1995): 63-86. *JSTOR*. Web. 23 June 2011.

Coover, Robert. "The Golden Age of Hypertext: The Passing of the Golden Age." Web. 15 Oct 2012.

--. *Universal Baseball Association, J. Henry Waugh, Prop*. New York: Random House, 1968. Print.

Coyne, Richard. *Designing Information Technology in the Postmodern Age*. Cambridge: MIT Press, 1995. Print.

Cowen, Robert. "Cottage Computing: Glorifying the Trivial?" *Technology Review* (Nov/Dec 1981): 6-7. Print.

--. "Home Computing Revisited." *Technology Review* (May/June 1982): 6-7. Print.

"Computers for the Masses ." *U.S. News & World Report* (27 Dec 1982): 64-72. Print.

Cummings, Robert E. "Coding with Power: Towards a Rhetoric of Computer Coding and Composition." *Computers and Composition* 23 (2006): 430-43. *ScienceDirect*. Web. 23 Oct 2012.

Curran, Lawrence J., and Richard S. Shuford. "IBM's Estridge ." *Byte* (Nov 1983): 88-97. Print.

Cusumano, Michael A., and David B. Yoffie. *Competing on Internet Time: Lessons from*

Microsoft and Its Battle with Netscape. New York: Free Press, 1998. Print.

Deleuze, Gilles and Felix Guattari. *Anti-Oedipus: Capitalism and Schizophrenia*. Trans. Robet Hurley, Mark Seem, and Helen R. Lane. New York: Viking, 1977. Print.

Despositio, Joe. "Computers: Which One Is For You?" *Popular Electronics* (May 1982): 45-60. Print.

Dijkstra, Edsger W. "GOTO Statement Considered Harmful." *Communications of the ACM*. 11.3 (1968): 147-8. *ACM Digital Library*. Web. 20 Aug 2013.

--. "The Humble Programmer." *Classics in Software Engineering*, Ed. Edward Nash Yourdon. New York: Yourdon Press, 1979. 113-28. Print.

--. "Notes on Structured Programming." *Structured Programming*, Eds. O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. New York: Academic Press, 1972. 1-81. Print.

Doctorow, Cory. "William Gibson Interview Transcript." Web. 21 Aug 2011.

Drucker, Johanna. "A Review of Matthew Kirschenbaum, *Mechanisms: New Media and the Forensic Imagination*." *Digital Humanities Quarterly* 3.2 (2009). Web. 6 March 2011.

Fallows, James. "Living With a Computer ." *The Atlantic Monthly* (Aug 1982): 84-91. Print.

Feuer, Lois. "The Calculus of Love and Nightmare: The Handmaid's Tale and the Dystopian Tradition." *Critique: Studies in Contemporary Fiction* 38.2 (1997): 83-95. *Taylor & Francis*. Web. 12 July 2013.

Freiberger, Paul and Michael Swaine. *A Fire in the Valley: The Making of the Personal Computer*. New York: McGraw-Hill, 2000. Print.

Fuller, Matthew. *Beyond the Blip: Essays on the Culture of Software*. Brooklyn: Autonomedia, 2003. Print.

--. *Media Ecologies: Materialist Energies in Art and Technoculture*. Cambridge: MIT Press,

2005. Print.

Galloway, Alexander R. *Protocol: How Control Exists After Decentralization*. Cambridge: MIT Press, 2004. Print.

Gamire, Elisa and Greg Pearson. *Tech Tally: Approaches to Assessing Technological Literacy*. Washingon D.C.: National Academies P, 2006. Print.

Gates, William. "A Trend Toward Softness ." *Creative Computing* (Nov 1984). Web. 17 Sept 2011.

Gens, Frank and Chris Christiansen. "Could 1,000,000 IBM PC Users Be Wrong?" *Byte* (Nov 1983): 135+. Print.

Geertz, Clifford. "Thick Description: Toward an Interpretive Theory of Culture." *The Interpretation of Cultures: Selected Essays*. 3–30. New York: Basic Books, 1973. Print.

Gibson, William. *Count Zero*. Ace Books, 1987. Print.

--. *Distrust that Particular Flavor*. New York: Berkley, 2012. Print.

--. *Mona Lisa Overdrive*. New York: Bantam Books, 1988. Print.

--. *Neuromancer*. New York: Ace Books, 1984. Print.

--. *Pattern Recognition*. New York: G.P. Putnum's Sons, 2003. Print.

Goldstone, Andrew, and Ted Underwood. "What can topic models of PMLA teach us about the history of literary scholarship?" *The Stone and the Shell*. Web. 20 Jan 2013.

Golumbia, David. *The Cultural Logic of Computation*. Cambridge: Harvard UP, 2009. Print.

Gorn, S. "Transparent-Mode Control Procedures for Data Communication, Using the American Code Standard for Information Exchange—A Tutorial ." *Communications of the ACM* 8.4 (1965): 203-6. *ACM Digital Library*. Web. 29 June 2012.

Grier, David Alan. *When Computers Were Human*. Princeton: Princeton UP, 2005. Print.

Hayles, N. Katherine. *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics.* Chicago: U of Chicago P, 1999. Print.

--. *My Mother Was a Computer: Digital Subjects and Literary Texts*. Chicago: U of Chicago P, 2005. Print.

--. *Writing Machines*. Cambridge: MIT Press, 2002. Print.

Heidegger, Martin. “The Question Concerning Technology.” *Martin Heidegger: Basic Writings from "Being and Time" (1927) to "The Task of Thinking" (1964),* Ed. David Farrell Krell. Harper: San Francisco. Print.

Hertzel, Leo J. “An Inteview With Robert Coover.” *Critique: Studies in Contemporary Fiction* 11.3 (1969): 25-9. *Proquest*. Web. 16 July 2013.

Hertzfeld, Andy. “Pirate Flag.” *Folklore.org*. Web. 23 Sept 2011.

--. “Signing Party .” *Folklore.org*. Web. 23 Sept 2011.

Hofmann, Thomas. "Probabilistic latent semantic indexing." *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. 1999. *Google Scholar*. Web. 7 July 2013.

Hugh, Thomas. “Protocols for Profit: Web and E-mail Technologies as Product and Infrastructure.” *The Internet and American Business*, Eds. William Aspray and Paul E. Ceruzzi. Cambridge: MIT Press, 2008. 105-158. Print.

Hutchins, Edwin L., James D. Hollan, and Donald A. Norman, “Direct Manipulation Interfaces.” *Human-Computer Interaction* 1 (1985): 311-88.

Jameson, Fredric. *Archaeologies of the Future: The Desire Called Utopia and Other Essays*. New York: Verso, 2005. Print.

Jockers, Matthew. *Macroanalysis: Digital Methods and Literary History*. Urbana: U of Illinois

P, 2013. E-text edition.

Kay, Alan. “Computer Software.” *Scientific American* 251.3 (1984): 53-59. Print.

--. “The Early History of Smalltalk.” Web. 14 Nov 2013.

--. “User Interface: A Personal View.” *The Art of Human-Computer Interface Design*, Ed. Brenda Laurel. Reading: Addison-Wesley, 1990. 191-207. Print.

Kirschenbaum, Matthew. “Editing the Interface: Textual Studies and First Generation Electronic Objects.” *Text* 14 (2002): 15-51. *JSTOR*. Web. 11 June 2013.

--. *Mechanisms: New Media and the Forensic Imagination*. Cambridge: MIT Press, 2008. Print.

Kittler, Friedrich A. *Gramophone, Film, Typewriter*, Trans. Geoffrey Winthrop-Young and Michael Wutz. Stanford: Stanford UP, 1999. Print.

Klarer, Mario. “Orality and Literacy as Gender-Supporting Structures in Margaret Atwood’s *The Handmaid’s Tale*.” *Mosaic* 28.4 (1995): 129-42. *ProQuest*. Web. 22 July 2013.

Kohl, Herbert. “Should You Buy Your Child a Computer?” *Saturday Evening Post* (Dec 1982): 34+. Print.

Krebs, Carl E., C. Bumgardner, and T. Northwood. “Terminal Transparent Display Language .” *AFIPS’76 Proceedings*. *Google Scholar*. Web. 29 June 2012.

Knuth, Donald and R.W. Floyrd. “Notes on Avoiding ‘Go To’ Statements.” *Information Processing Letters* 1 (1971): 23-31. *ScienceDirect*. Web. 21 Aug 2013.

Kuhn, Adrian, et al. “Semantic Clustering: Identifying Topics in Source Code.” *Information and Software Technology* 49.3 (2007): 230-43. *ScienceDirect*. Web. 14 Aug 2012.

Lantz, Keith A., and Richard F. Rashid. “Virtual Terminal Management in a Multiple Process Environment .” *SOSP ’79 Proceedings*. *Google Scholar*. Web. 29 June 2012.

Landow, George P. *Hypertext 3.0: Critical Theory and New Media in an Era of Globalization*.

Baltimore: Johns Hopkins UP, 2006. Print.

Latour, Bruno. *Reassembling the Social: An Introduction to Actor-Network Theory*. New York: Oxford UP, 2005. Print.

Liu, Alain. "The Meaning of the Digital Humanities." *PMLA* 128 (2013): 409-23. Print.

Liu, Alain, et al. "Born-Again Bits: A Framework for Migrating Electronic Literature." *Electronic Literature Organization*. Web. 13 Oct 2012.

Lemmons, Phil. "An Interview with the Macintosh Design Team." *Byte* (Feb 1984): 58+. Print.

--. "Patronizing the Naïve User." *Byte* (July 1984): 6. Print.

--. "What Makes PCs Special." *Byte* (Aug 1984): 6. Print.

Leonard, Andrew. "Nodal Point." *Salon.com*. Web. 23 June 2011.

Lessig, Lawrence. *Code, And Other Laws of Cyberspace: Version 2.0*. New York: Basic Books, 2006. Print.

Levy, Steven. *Hackers: Heroes of the Computer Revolution*. Garden City: Anchor, 1984. Print.

Lippman, Walter. *Public Opinion*. New York: Macmillan, 1922. Print.

Lukins, Stacy K. "Bug Localization Using Latent Dirichlet Allocation." *Information and Software Technology* 52.9 (2010): 972-90. 14 Aug 2012.

Lunenfeld, Peter, et al. *Digital_Humanities*. Cambridge: MIT Press, 2012. Print.

Lundqvist, Andreas, et al. "GNU/Linux Distribution Timeline." Web. 8 Oct 2012.

Mackenize, Adrian. *Cutting Code: Software and Sociality*. New York: Peter Lang, 2006. Print.

Mahoney, Michael. "The History of Computing in the History of Technology." *Annals of the History of Computing* 10.2 (1988): 113-25. *IEEExplore*. Web. 5 Feb 2014.

Manovich, Lev. *The Language of New Media*. Cambridge: MIT Press, 2001. Print.

Markley, Robert. *Dying Planet: Mars in Science and the Imagination*. Durham, Duke UP, 2005.

Print.

--. "Boundaries: Mathematics, Alienation, and the Metaphysics of Cyberspace." *Virtual Realities and Their Discontents*, Ed. Robert Markley. Baltimore: Johns Hopkins UP, 1996. Print.

Marino, Mark C. "Critical Code Studies." *Electronic Book Review*. Web. 4 March 2011.

McCafferty, Larry. "An Interview With William Gibson." *Mississippi Review* 16.2 (1988): 217-36. *JSTOR*. Web. 23 June 2011.

--. Ed. *Storming the Reality Studio: A Casebook of Cyberpunk and Postmodern Science Fiction*. Durham: Duke UP, 1991. Print.

McCallum, E.L. "Mapping the Real in Cyberfiction ." *Poetics Today* 21.2 (2000): 349-77. *EBSCO*. Web. 23 June 2011.

McCarty, Willard. *Humanities Computing*. Basingstoke: Palgrave Macmillan, 2005. Print.

McDonald, Kyle. "When Art, Apple and the Secret Service Collide: 'People Staring at Computers.'" *Wired.com*. Web. 13 July 2012.

McGann, Jerome. *A Critique of Modern Textual Criticism*. Chicago: U of Chicago P, 1983. Print.

--. *Radiant Textuality: Literature After the World Wide Web*. New York: Palgrave, 2001. Print.

McPherson, Tara. "U.S. Operating Systems at Mid-Century: The Intertwining of Race and UNIX." *Race After the Internet*, Eds. Lisa Nakamura and Peter Chow-White. Milton Park, Routledge, 2012. 21-37. Print.

Mirowski, Philip. *Machine Dreams: Economics Becomes a Cyborg Science*. New York: Cambridge UP, 2002. Print.

Montfort, Nick. *Twisty Little Passages: An Approach to Interactive Fiction*. Cambridge: MIT Press, 2003. Print.

Montfort, Nick and Ian Bogost. *Racing the Beam: The Atari Video Computer System*.

Cambridge: MIT Press, 2009. Print.

Montfort, Nick and Noah Wardrip-Fruin. “Acid-Free Bits: Recommendations for Long-Lasting Electronic Literature.” *Electronic Literature Organization*. Web. 13 Oct 2012.

Montfort, Nick, et al. *10 PRINT CHR$(205.5+RND(1));:GOTO 10*. Cambridge: MIT Press, 2013. Print.

Naur, Peter and Brian Randell, Eds. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmsich, Germany, 7th to 11th October, 1968.* Brussels: Scientific Affairs Division, NATO. 1969. Print.

Neisser, Ulrich. *Cognitive Psychology*. Upper Saddle River: Prentice Hall, 1967. Print.

--. “The Imitation of Man by Machine.” *Science* 18 (1963): 193-97. *AAAS*. Web. 8 July 2012.

Nelson, Theodor. *Computer Lib / Dream Machines*. Redmond: Tempus Books, 1987. Print.

Nixon, Nicola. “Cyberpunk: Preparing the Ground for Revolution or Keeping the Boys Satisfied?” *Science Fiction Studies* 19.2 (1992): 219-35. *JSTOR*. Web. 23 June 2011.

Norman, Donald A. “Cognitive Artifacts.” *Designing Interaction: Psychology as the Human-Computer Interface*. Ed. John M. Carroll. New York: Cambridge, 1991. 17-38. Print.

--. *The Design of Everyday Things*. New York: Basic Books, 1988. Print.

--. “The Problem With Interfaces.” *The Art of Human-Computer Interface Design*, Ed. Brenda Laurel. Reading: Addison-Wesley, 1990. 208-20. Print.

Orth, Maureen. “For Whom Ma Bell Tolls Not.” *Los Angeles Times*. Oct 31, 1971. *LexusNexus*. Web. 10 Nov 2009.

Peddle, Chuck. “Counterculture to Madison Avenue.” *Creative Computing* (Nov 1984). Web. 17 Sept 2011.

Poole, Steven. “Tomorrow’s Man.” *Guardian* (3 May 2003). Web. 21 Aug 2011.

Porush, David. “Cybernetic Fiction and Postmodern Science.” *New Literary History* 20.2 (1989): 373-96. *JSTOR*. Web. 23 June 2011.

Pournelle, Jerry. “Are We Dinosaurs?” *Bye* (Nov 1984): 361+. Print.

--. “The Chaos Manor Gets Its Long Awaited IBM PC.” *Byte* (Feb 1984): 113-6. Print.

Punday, Daniel. “Creative Accounting: Role-playing Games, Possible-World Theory, and the Agency of Imagination.” *Poetics Today* 26.1 (2005): 113-39. *EBSCO*. Web. 16 July 2013.

Rheingold, Howard. *Virtual Reality*. New York: Summit Books, 1991. Print.

Rogers, Michael and Jennet Conant. “It’s the Apple of His Eye.” *Newsweek* (30 Jan 1984): 54-7. Print.

Rosenbaum, Ron. “Secrets of the Little Blue Box.” *Esquire* Oct 1971. *EBSCO*. Web. 10 Nov 2009.

Rotman, Brian. *Ad Infinitum: The Ghost in Turing’s Machine*. Stanford: Stanford UP, 1993. Print.

Rubin, Charles. “Macintosh: Apple’s Power New Computer .” *Personal Computing* (Feb 1984): 56+. Print.

Rubin, Charles and Kevin Strehlo. “Why So Many Computers Look Like The ‘IBM Standard,’” *Byte* (Mar 1984): 52+. Print.

Rubin, Frank. “’GOTO Considerered Harmful’ Considered Harmful.” *Communications of the ACM* 30.3 (1987): 195-6. *ACM Digital Library*. Web. 20 Aug 2013.

Rush, Anna Fisher. “Does a Home Computer Make Sense for You?” *McCall’s* (Sept 1982): 62+. Print.

Sag, Matthew. “Orphan Works as Grist for the Data Mill.” *Berkeley Technology Law Journal*. 30

Aug 2012. *Social Science Research Network*. Web. 1 July 2013.

Sample, Mark L. “Unseen and Unremarked On: Don DeLillo and the Failure of the Digital Humanities.” *Debates in the Digital Humanities*. Ed, Matthew K. Gold. Minneapolis: U of Minnesota P, 2012. Web. 15 Oct 2012.

Schiffers, Manuel. “Behind the Shakeout in Personal Computers.” *U.S. News & World Report* (27 June 1983): 59-60. Print.

Shneiderman, Ben “Direct Manipulation: A Step Beyond Programming Languages.” *Computer* 16. 8 (1983): 57-69. *IEEEXplore*. Web. 25 Jun 2012.

--. *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge: Winthrop, 1980. Print.

Siivonen, Timo. “Cyborgs and Generic Oxymorons: The Body and Technology in William Gibson’s Cyberspace .” *Science Fiction Studies* 23.2 (1996): 227-244. *JSTOR*. Web. 23 June 2011.

Simon, Herbert A. “A Behavioral Model of Rational Choice .” *Quarterly Journal of Economics* 69 (1955): 99-118. *Science Direct*. Web. 16 Feb 2014.

--. “The Organization of Complex Systems .” *Hierarchy Theory: The Challenge of Complex Systems*, Ed. H. H. Patee. New York: G. Braziller, 1973. Print.

--. “Rational Choice and the Structure of Environment .” *Psychological Review* 63 (1956): 129-38. *ScienceDirect*. Web. 16 Feb 2014.

Sneddon, Joey-Elijah. “Blogger Claims Ubuntu’s New Shopping Lens Breaks EU Law.” *OMG Ubuntu!* Web. 4 Aug 2012.

Stallman, Richard M. *Free Software, Free Society: The Selected Essays of Richard M. Stallman*. Ed. Joshua Gay. Boston: Free Software Foundation, 2002. Print.

Stephenson, Neal. *In the Beginning…Was the Command Line*. New York: Avon Books, 1999. Print.

Sterling, Bruce. “Cyberpunk in the Nineties.” *Interzone*. Web. 17 Aug 2011.

--. “Dead Media: A Modest Proposal and Public Appeal.” *Well.com*. Web. 14 Sept 2011.

--. “Get the Bomb Off My Back.” *New York Times* (13 Oct 1991). *LexisNexus*. Web. 3 Aug 2011.

--. “The Life and Death of Media.” Speech. Web. 14 Sept 2011.

--. Ed. *Mirrorshades: The Cyberpunk Anthology*. New York: Arbor House, 1986. Print.

--. “State of the World 2013.” *Well.com*. Web. 26 May 2013.

“Steve Jobs and the Future of Apple Computer.” *Personal Computing* (April 1984): 242-48. Print.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Reading: Addison-Wesley, 1994. Print.

Suchman, Lucy. *Plans and Situated Actions: The Problem of Human-Machine Communication*. New York: Cambridge, 1987. Print.

Tanenbaum, Andrew S. *Modern Operating Systems*. 3rd Edition. Upper Saddle River: Pearson Prentice Hall, 2008. Print.

--. *Structured Computer Organization.* 5th Edition. Upper Saddle River: Pearson Prentice Hall, 2006. Print.

Tanenbaum, Andrew S. and Robbert Van Renesse. “Distributed Operating Systems .” *Computing Surveys* 17.4 (1985): 419-70. *ACM Digital Library*. Web. 29 June 2012.

Thomas, Stephen W., et al. "Modeling the evolution of topics in source code histories." *Proceedings of the 8th working conference on mining software repositories*. 2011. *ACM Digital Library*. Web. 8 May 2013.

Torvalds, Linus and David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. New York: Harper Business, 2001. Print.

Turing, Alan. "Intelligent Machinery." *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma*, Ed. B. Jack Copeland. New York: Oxford UP, 2004. 395-432. Print.

--. "On Computable Numbers, with an Application to the Entscheidungs Problem." *The Essential Turing*. 91-124. Print.

Turkle, Sherry. *Life on the Screen: Identity in the Age of the Internet*. New York: Simon & Schuster, 1995. Print.

--. *The Second Self: Computers and the Human Spirit*. New York: Simon & Schuster, 1984. Print.

Turner, Fred. *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*. Chicago: U of Chicago P, 2006. Print.

Uttal, Bro. "Sudden Shake-Up in Home Computers ." *Fortune* (11 July 1983): 105-6. Print.

Wardrip-Fruin, Noah. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. Cambridge: MIT Press, 2009. Print.

Weizenbaum, Joseph. *Computer Power and Human Reason: From Judgment to Calculation*. San Francisco: W.H. Freeman, 1976. Print.

Whalen, Terence. "The Future of Commodity: Notes Towards a Critique of Cyberpunk and the Information Age ." *Science Fiction Studies* 19.1 (1992): 75-88. *JSTOR*. Web. 23 June 2011.

Wiener, Norbert. *The Human Use of Human Beings: Cybernetics and Society*. 2nd Ed. New York: Avon Books, 1967. Print.

Williams, Gregg. “A Closer Look at the IBM Personal Computer.” *Byte* (Jan 1982): 36+. Print.

--. “The Apple Macintosh Computer.” *Byte* (Feb 1984): 30+. Print.

Winograd, Terry and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Reading: Addison-Wesley, 1987. Print.