

2 Software and Ideology

Provides classes and interfaces for obtaining reflective information about classes and objects.

*Description of the Java programming package
“java.lang.reflect,” Sun Microsystems (2004).*

An Analog . . .

In the previous chapter I described the interface as a general mode of mediation. While readily evident in things like screens and surfaces, the interface is ultimately something beyond the screen. It has only a superficial relationship to the surfaces of digital devices, those skins that beg to be touched. Rather, the interface is a general technique of mediation evident at all levels; indeed it facilitates the way of thinking that tends to pitch things in terms of “levels” or “layers” in the first place. These levels, these many interfaces, are the subject of analysis not so much to explain what they are, but to show that the social field itself constitutes a grand interface, an interface between subject and world, between surface and source, and between critique and the objects of criticism. Hence the interface is above all an allegorical device that will help us gain some perspective on culture in the age of information. For this reason, we look now to the “deeper” realm of software, the realm below the screen, with an eye to the possible ideological construction of this hidden electronic kingdom.

New media, and the software that drive it, has always had an interesting relationship with ideology. Some see new media as a liberating sword, cutting through the ideological fog of the old world, while others see new media as another apparatus of control, insinuating itself into every corner of our lives. For this reason I want to structure the present chapter around a particular essay, Wendy Hui Kyong Chun's fascinating "On Software, or the Persistence of Visual Knowledge," in which the claim is made, among other things, that "software is a functional analog to ideology." The two moments contained in this claim – "an analog" "that is functional" – will help to structure the chapter. As brief as it is, Chun's claim nevertheless brings out a rich set of discussions, not least of which are the theory and history of both software and ideology, the question of *functionality*, and also the way in which something might be an analog for something else. Further, this last matter, the analog, is particularly apropos for an exploration of the process of interpretation and how cultural objects deploy modes of figuration such as allegory (an issue to which I will return at the end of the chapter). While the question of functionality will occupy the second half of this chapter it is worth defining now: I adopt the usage from computer science where a function is any subroutine or isolated block of code, but further from mathematics where a function is any expressive entity able to translate a set of inputs into a set of outputs. One of the first claims extracted from Chun is that there is a similarity, which we may call a formal similarity, between the structure of ideology and the structure of software. Conveniently, the desire to make such a claim can be folded into the quality of software itself: it is a technology of simulation and thus has as a core asset the ability to "take shape" in various ways. In other words, software is by definition formal (as symbolic or abstract mathematical and logical code), and thus it acclimates well to structural comparisons – even better than, one might argue, its cousins the visual image or verbal narrative. The analogical or "expressive" theory of ideology is also not unfamiliar, as in the work of Louis Althusser where the structure of ideology resembles, more or less, an architectural drawing of a house, with the material base of society down

below and the cultural or superstructural layer up above. Ideology emerges not strictly as the house itself, but as a figurative projection of one layer onto the other. It is thus a stand in for what the Marxists simply call *history*, or in crass terms the ongoing reification of social relations and processes, and further the problem of being able to represent these details from the past in the present. If software is less a vehicle for ideology and more its simulation or model, it is no doubt because of these formal qualities of software which, combined with software's importance as so many cultural and historical artifacts, are so well suited for precise mathematical modeling of real world phenomena – in functionality, in spatial projection, in sight and sound – but in a manner in which the very material distance or empirical falsity of that simulation is at once foregrounded as a fatal flaw and then resolved as insignificant. “Ideology” is one name that may be used for things that act this way.

Gaps abound in such systems. And across these many gaps an intense mimetic thicket emerges: ideology as projection across the gap between individuals and their real conditions of existence; software as mimetic technology bridging the gap between machines and their logical simulation; and the startling claim, from Chun, that software could conceivably be a continuously variable copy (i.e. an analog) of ideology.

It would be logical to begin such a discussion by revisiting the classical theories of ideology, which I will do quickly now, ignoring for a moment that ideology and its own conceptualization in theoretical discourse seem to be at odds, for the power of ideology, if it has any at all, is the high level of constraint it puts on discourse itself, aiming for a system of total or “smooth” thought. Ideology is always about two things at the same time. First, ideology is a matter of life and culture, a representation of one's lived social relation (following Althusser's definition). But second, ideology is also implicitly a matter of *critique*, for as Jameson said once about capitalism, simply to utter the word “ideology” seems to indicate one's disdain for it. In discourse on the subject, a first motif, that of *scientific thought*, suggests that ideology is always set up in distinction to other styles of thinking. Yet the history of the

concept takes a detour here, for whereas Destutt de Tracy, who coined the term in the 1790s, championed his new science of ideology as an antidote to philosophy and metaphysics (and later was famously lambasted by Napoleon for caring little about the “knowledge of the human heart” and “the lessons of history”), Althusser, by contrast, reverses the sophistry narrative, positing Marxist critique as scientific and ideology as the threat to clear thinking.

Running a close second to the scientific narrative is the *narrative of political failure*. Here, ideology serves as a sort of anthropological “explanation” for all sorts of alienation, exploitation, and false consciousness existing in society, along the lines of the doctrine of original sin or the great theological arguments for the existence of evil as explanation for why the here and now is so inferior to the hereafter. This is an essentially strategic theory of ideology, evident in Gramsci and others, that serves as theoretical proof for the nonexistence of the revolution, despite the perception that all the necessary real world phenomenon are in place for it to arrive at any moment. (This same political messianism is addressed quite compellingly in Derrida’s *The Specters of Marx*).

Ultimately these first two narratives are synthesized into a third, ideology’s *determinism narrative* in which ideology is understood as a system of total thought that may or may not determine things like human subjects. Here we may include the work of the Frankfurt School, or again Althusser, with the key issue being the industrialization of the body and mind into ever more insidious modes of efficiency and instrumentality. Likewise we might return here to Marx and Engels, particularly the so-called “dominant ideology thesis” extracted from passages in the *Communist Manifesto* as well as certain references to the determination of consciousness in *The German Ideology*.

Marx’s theory of ideology, such as it is, has long been critiqued for its determinism. But the theory of ideology was recuperated in the later part of the twentieth century by what might be thought of as the *synthetic narrative*: ideology is not a one-way street, but is always meted out at the intersection of cultural production and cultural consumption. In this context,

ideology follows a more dialectical logic, as with Jameson's two terms "reification and utopia" (a formulation from *Signatures of the Visible*, the terms of which shift slightly to the "utopia and ideology" of *The Political Unconscious*), or with Stuart Hall's well-known theory of "articulation," or indeed earlier with Gramsci's notion of hegemony in which political or cultural dominance is always the result of active negotiation and production.

Ideology is not something that can be solved like a puzzle, or cured like a disease. Even with all these various and sometimes conflicting themes from the many theories of ideology, it would be shortsighted to write off the concept as some sort of cognitive delusion, a fog of false consciousness afflicting the minds of those it touches. Instead ideology is better understood as a problematic, that is to say a conceptual interface in which theoretical problems arise and are generated and sustained precisely as problems in themselves. Following Chun's lead, it appears that software too must be understood not as a given social and technical object, but as a problematic interface – indeed, one that is continuously in the process of producing its own status as social and technical. (In this sense "problem" is a synonym for "interface," which is itself a stand-in candidate for "software.")

Most discussions of software require a significant amount of back-peddling at the outset, owing to a number of confusions and difficulties with the concept. The first difficulty is that software, in different ways and in different amounts, customarily stands in opposition to the notion of the qualitative or the continuous, often summed up as the "realm of the analog." (Although the term analog is so often misused it should be uttered with extreme care.) While this analog-digital polarity is thorny in itself, we must be particularly attentive given the current discussion, as the topic of the analogical has already been broached in Chun's description of the internal modeling of software as something like ideology-in-code. Second, software relies on the assumption that there is something like a programmer and something like a user. This also presents a special set of problems, the most important of which is the status of the actor versus the acted-upon (the

aforementioned determinism narrative), and under what circumstances which is which. Today the “culture industry” takes on a whole new meaning, for inside software the “cultural” and the “industrial” are coterminous (which is why it is tautological to speak today of a cultural logic of informatics). The detailed forms of algorithmic interaction and play required today of the computer-using public is, in my mind, so exactly akin to writing code that the division between the two must certainly be ascribed to other ends. Perhaps it has to do with the creation and maintenance of another class of priest-like specialists striving to decode software just as the critic longs to “reverse engineer” ideology, or perhaps it is best to contextualize it within the long standing debates around groups of producers and groups of consumers, and the implicit power dynamics haunting their mutual distinction. A third difficulty is the notion that, in using the concept of software, one is somehow excluding hardware. This flows from a long-standing assumption that data is fundamentally immaterial or ethereal and that, conversely, machines are the stuff of material cogs and levers. It is a foundational claim stemming from the very first informatic machines. But as Chun and others have pointed out, “programming” a computer originally meant patching circuits together using cables or connectors and thus “software” began historically not as executable software applications as we know them today but as any sort of service labor performed in or on informatic machines; even video was once known as “software,” to distinguish it from “hard” playback decks and cameras. Thus the interface effect is also a kind of segregation effect whereby data is relegated to the realm of ideas and machines to the realm of technology. This results in a presentism, admittedly parroted by me here, where software is understood only in its late-twentieth-century definition as a symbolic machine language and not in an earlier definition in which software might rightly be understood as pre- or non-linguistic.

Unraveling such logic is challenging. Friedrich Kittler tried to tackle it in his essay “There is No Software,” which I already referenced briefly in the introduction. The essay is a clever if sometimes casual unpacking of the strict division placed

between code and machines. The conceit of the essay's title is that, indeed, software is merely a human-friendly category extracted from what is always an operation of hardware. Logic gates are electronic machines; they are physical devices through and through. Voltages in electronic circuits are material, not immaterial (whatever that may be). As it was in Parmenides: what is not, cannot be. So one must assume that the "soft" comes from the informatic fluidity of these devices, from what Turing meant when he called his machine "universal" rather than "discrete." When basic logic gate functionality is abstracted and strung together into machine commands, translated into assembly op-codes, and then later articulated in a higher level computer language such as C, the argument from Kittler is that one should never understand this "higher" symbolic machine as anything empirically different from the "lower" symbolic interactions of voltages through logic gates. They are complex aggregates yes, but it is foolish to think that writing an "if-then" control structure in eight lines of assembly code is any more or less machinic than doing it in one line of C, just as the same quadratic equation may swell with any number of multipliers and still remain balanced. The relationship between the two is *technical*.

Still its being technical does not excuse it from being *interesting*, particularly on the question of ideology. It simply requires that we speak in terms of a machine aesthetics, rather than a verbal or visual aesthetics. What I shall propose here is that *software is an example of technical transcoding, without figuration, that nevertheless coexists with an exceedingly high level of ideological fetishism and misrecognition*. (Is this not, after all, the very definition of technology?) Chun makes this connection explicitly: "Software is based on a fetishistic logic. Users know very well that their folders and desktops are not really folders and desktops, but they treat them as if they were – by referring to them as folders and desktops."² Whether this is truly a fetishistic logic, or an allegory for one, remains to be determined. In Marx, fetishism comes from the expressive and figurative logic of representation – how value appears in the form of something that it isn't – a fact no doubt that allowed a Marxist methodology to translate easily to other

intellectual fields also dealing with the problem of representation (semiotics, art history, literary criticism, theories of race, and so on). But of course the strength of Marx's analysis in *Capital* is that he derived fetishism from a fundamentally empirical, or "technical," set of relations (the rule of market exchange, the standardization of labor power, the sciences of productivity and efficiency, the operation of machines, and so on). Thus a dialectic of technical transcoding and fetishistic abstraction exists from the start. This is why I will suggest at the end of the chapter that the relationship between software and ideology is best understood as an allegorical one: software is not merely a vehicle for ideology; instead, the ideological contradictions of technical transcoding and fetishistic abstraction are enacted and "resolved" within the very form of software itself.

New media hide as much as they show, and the ultimate power of Chun's essay is found in a symptom, an idea that appears in the title but then hides itself during most of her essay just as fiercely as it proclaims itself at the start. For Chun the interface between software and ideology is a throwback: it is *visual* knowledge that persists inside software. Thus one might assume that the visual quality of knowledge is the key to the software/ideology puzzle. "The computer," she writes, "that most nonvisual and nontransparent device – has paradoxically fostered 'visual culture' and 'transparency.'"³ Software purists will doubtless be put off by this, as will those more familiar with the intellectual terrain of visual culture, to which the topics of my writings, admittedly, are only related through a sort of counterintuitive leap. If I may read between the lines of Chun's essay, it is not exactly the discipline of visual culture that provides a backdrop for her project, despite the use of the term "visual" and despite indications made in her book *Control and Freedom*. And here things start to get thorny, for one must make a distinction between the "visible," which is typically understood as specific to the faculty of optical sight, and the "visual," which might be understood in broader, more figurative strokes as an epistemic process of cognitive understanding and conceptualization: one may speak of mental "insight" with or without the optical faculty, just as one

might “see” an image inside of a dream. This results in the sorts of claims, astounding at first blush, made by W.J.T. Mitchell most recently in *What Do Pictures Want?* that the core, genetic formation of culture is not the text or even the idea, but the *image*.

Of course, to read Mitchell (or Chun) sympathetically would not be to assume he believes that the world is made up exclusively of semiotic interfaces like television screens, advertisements, paintings, film reels, and corporate logos – no, these are not the “images” he means, at least not only. Instead, the “visual” might better be understood as referring to any likeness or motif that fixes some grouping of elements, such that one might “see” these elements both as a relational whole (as a memory, refrain, gesture, raster, etc.) and also in terms of their constituent parts (phoneme, texture, color, pitch, pixel). This is not such a dramatic claim, however, and it is certainly one that runs parallel with Western philosophy and aesthetics from the get go. Thus, the enlightenment episteme, which unites (some might say collapses) knowledge and the visual in various technologies of representational transparency and communicability, persists in software, argues Chun, not only because of the conceptualizations and “sightings” just mentioned, but also to the extent that it promotes a depth model of representation between sources and surfaces, scripts and screens, the code and the user. But the enlightenment model has also provoked a whole flock of criticism, rightly, around the impossibility of such transparent representation. Thus we arrive at a paradox: any mediating technology is obliged to erase itself to the highest degree possible in the name of unfettered communication, but in so doing it proves its own virtuosic presence as technology thereby undoing the original erasure. “What is software,” Chun writes, “if not the very effort of making something explicit, or making something intangible visible, while at the same time rendering the visible (such as the machine) invisible?”⁴ Language wants to be overlooked. But it wants to be overlooked precisely so that it can more effectively “over look,” that is, so that it can better function as a syntactic and semantic system designed to specify and articulate while remaining detached from the very processes of

specificity and articulation. This is one sense in which language, which itself is not necessarily connected to optical sight, can nevertheless be “visual.”

Chun’s suggestion is that to understand software we must return to a discussion of the visual and the ideology problems contained therein, not skip forward to some purely machinic, and hence chiefly nonvisual, aesthetic realm. This is good advice, but the appeal to visual knowledge must still be understood in a figurative, not literal (i.e. optical), sense, for as I will argue below it is more valuable to separate rather than collapse software’s visual and machinic aspects in mutually distinct struggle, for this separation simulates the same struggle writ large in the socio-political arena between an informatic or machinic model of organization and a slightly older one which takes as its prized aesthetic forms of the verbal narrative and the visible image. In other words, the separation between the visual and the machinic in software is important because it is an allegory of the social. Chun’s “persistence of visual knowledge” signifies the double bind of what some optimistically call the information age: it underscores the fact that software is rooted in symbolic logic not optical vision, and thus cannot fully leverage the dominant form existing even now in the spectacle society, yet at the same time gains inroads precisely at the expense of that social form that it so effectively simulates.

The history of visuality and computing is a complicated history. It is certainly incorrect to divorce one from the other, as authors like Lev Manovich have rightly pointed out (see in particular his essay “The Automation of Sight”). Indeed any understanding of contemporary visual mediation that ignores software does so at its own peril, in an age when cinema has become synonymous with Final Cut Pro, photography with Photoshop, writing with Microsoft Word, and on and on. The history of the pixel is instructive in this capacity: at its invention in the middle twentieth century the electronic pixel was essentially the same thing as the binary bit, one existing in the modality of visible light and the other existing in the modality of mathematical value. But at the same time I am sympathetic to a certain minoritarian refrain running through recent media

theory on the specificity of computers as non-optical if not altogether non-visual media, for anyone wishing to cram computers into the framework of “visual culture” is certainly suffering from an unfortunate fetishization of the physical interface, as if the computer monitor were an adequate substitute for the medium as a whole, which, in addition to screens of various shapes and sizes, consists of any number of other technologies: nonoptical interfaces (keyboard, mouse, controller, sensor); data in memory and data on disk; executable algorithms; networking technologies and protocols; and the list continues. The fields of computer vision and computer graphics are also but a fraction of computer science as a whole which occupies most all of its time with algorithms, data structures, cryptography, robotics, bioinformatics, networking, machine learning, and other nonvisual applications of symbolic systems.⁵

Each of these domains deserves a deeper level of attention. But while leaving this discussion of the visual somewhat unresolved, I return to the overarching theme, the interface, by offering the first of two general observations on software and ideology. Software operates through a technological model that places a great premium on meticulous symbolic declarations and descriptions, yet at the same time requires concealment, encapsulation, and obfuscation of large portions of code. This is why programmers talk in terms of “software interfaces” or “application interfaces.” Formulated as an assertion, *software requires both reflection and obfuscation*. If software is indeed an allegorical analog to ideology, it should come as no surprise that software functions in such a dialectical fashion. The critics of ideology have often described it in synthetic terms (Jameson, Hall, Gramsci, et al.). But software has its own technologies of reflection and obfuscation. *Reflection* is nearly axiomatic: the complete syntactic and semantic rules of a computer language must be defined and written into any environment designed to interpret, parse, or execute it. (As an aside: in the so-called natural languages this is never the case, despite style guides and dictionaries, as unforeseen “inductive” uses of language may be stumbled upon or invented without the blessing of provenance, whereas with software the

unforeseen articulations of language are essentially dismissed out of hand as errors or “exceptions.” Of course, this does not foreclose on the possibility, nay necessity, for hacks and other software exploits to pop up in the complexity of the software network, exploits which can never be predicted, as the computer scientists would say, “statically.” The difference is that exploits operate *intensively* within and through the rules of the symbolic system, while natural languages operate *extensively* as a result of a combinatorial discursive logic ever intent on probing the boundaries of allowable style.) Reflective sandboxing of software code within a machine built to parse it is seen in the case of a computer language like Java which must be compiled and then run as bytecode inside a special runtime environment, or, as with the language C, compiled and then run as “native” machine instructions, or with a simple mark up language like HTML the specifications for which must be entirely designed into any browser destined to interpret and display it, or also with other interpreted code such as a three-dimensional model whose mathematical values for vertices and textures must be transcoded according to the rules of a given data format and a given style of visual projection. This existential, or meta-medial reflection is further illustrated in the “system” or “global” paratextual variables existent in many languages. Perl’s implementation is particularly dazzling: \$\$ for the process number, \$! for the last system call error, and \$0 for the program name; Java too, which includes special “meta” objects such as the package `java.lang.reflect` and the class `java.lang.Class` that are designed to obtain information about classes and objects.⁶ These meta objects are used to write reflexive software such as debuggers or interpreters, or to declare new objects dynamically during runtime. Indeed ontology itself, formerly a branch of philosophy, is now also a branch of computer science appearing perhaps most visibly in Web ontology standards such as OWL (Web Ontology Language).

Encapsulation and *transcoding* are two useful ways for understanding the technology of obfuscation. The principle of transcoding, which I am adopting from Manovich, states that new media objects may be converted digitally from one data

structure to another, but further that there are entire media formats based entirely on such conversions and nothing more. Thus an application like BIND (Berkeley Internet Name Domain), a leading domain name resolver, exists so that IP addresses can be masked by more human-legible domain names. The netmask is similar: in binary math, a bit within a binary number can be extracted using a special masking number and an “and” bitwise operation; likewise in network addressing, subnets are defined using a special number called a netmask that specifies a section of the network address for the subnet itself and then “masks” or obfuscates the rest to be used by the hosts residing on the subnet. In a bitmap image numerical values are used to represent color intensities in a pixel grid. Yet they are never represented as such, but instead are converted into data signals and sent to the display adapter which then converts these values into voltages that appear as light on a screen (the aforementioned modal transformation from bit to pixel). So in this case “conversion” is a certain conjunction between “physical” signal and “abstract” number where one is hidden at the expense of the other. Even the HTML example referenced previously uses the same principle of data hiding: HTML is never shown in the browser window, it is always parsed and converted from ASCII text into a graphical layout (which may or may not also include ASCII text).

These are only a few examples of the larger trend in software design to hide numerical encoding of data behind more privileged “semantic” formats such as natural language or graphics. In this way, *numbers essentially follow an occult logic*: they are hidden at exactly the moment when they express themselves. Chun calls this “the nonreflection of changeable facts in software.”⁷ In fact encapsulation is rather dominant in the area of code authorship and compilation. The most fundamental design principle for object-oriented computer languages is the combination of variables and operations on variables (methods) into something called a class. Classes can be instantiated as objects and these objects interact through the ability to send messages to other objects via object interfaces. This is where encapsulation comes in: the details of how

an object implements any given operation are deliberately kept hidden from any other object making requests of it. For example, a method's input and output might be visible but how it processes the input into the output is kept hidden. In order to implement the technologies of encapsulation, a system of visibility modifiers are employed, as in the case of Java whereby classes, methods, or variables may be deemed "public," "private," or "protected," each designation helping to determine if the class, method, or variable is visible to the rest of the code.

Code obfuscation, or "information hiding," is employed in order to make code more modular and abstract and thus easier to maintain. A class' method can be updated and, as long as it continues to fit its public interface "signature," one may be reasonably assured the code will continue to run. The following text articulates the rationale for obfuscation from the perspective of computer science.

A major challenge – perhaps *the* major challenge – in the construction of any large body of software is how to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously. This modularization of efforts depends critically on the notion of *information hiding*, which makes objects and algorithms invisible, whenever possible, to portions of the system that do not need them. Properly modularized code reduces the "cognitive load" on the programmer by minimizing the amount of information required to understand any given portion of the system. In a well-designed program the interfaces between modules are as "narrow" (i.e., simple) as possible, and any design decision that is likely to change is hidden inside a single module. This latter point is crucial, since maintenance (bug fixes and enhancement) consumes many more programmer years than does initial construction for most commercial software.

In addition to reducing cognitive load, information hiding has several more pedestrian benefits. First, it reduces the risk of name conflicts: with fewer visible names, there is less chance that a newly introduced name will be the same as one already in use. Second, it safeguards the integrity of data abstractions: any attempt to access objects outside of the subroutine(s) to which they belong will cause the compiler to issue an "undefined

symbol” error message. Third, it helps to compartmentalize runtime errors: if a variable takes on an unexpected value, we can generally be sure that the code that modified it is in the variable’s scope.⁸

On this point it pays to be overly literal. It is worth spelling out the significant similarity between such a description of the labor process and that same description offered by Marx and his progeny in the ideology discussion. In both cases we have an “object” imbued with a complex interface for hiding things, be it the commodity object (or as Guy Debord maniacally demonstrated, the commodity as image) and its ability to mask its own history of production and the social division of labor that generated it, or be it the Java object and its ability to cordon off various functionality into this or that site of inscription and execution, which is no doubt an abstraction or *mapping* of the actual division of labor globally in the dot-com firm producing it, where one part of the code might spring from a desk in Redmond and another part from a desk in Bangalore without anyone being the wiser. The structure of software facilitates this larger social reality. This is not to promote some sort of conspiracy theory for the new economy, simply to note the significant formal similarities between the structure of software as a media technology and the structure of ideology – with the commodity as a waypoint between the two. (Recall that for figures like Debord, Jean-Joseph Goux, or Roland Barthes the commodity and ideology are nearly synonymous.)

New media is often lauded for championing the virtue of openness. Yet I stress that “reflection” and “obfuscation” have nothing to do with the debate around open source versus proprietary software. They have nothing to do with “good” uses of code versus “bad” uses. Open source software follows the principle of source concealment, just as proprietary software does. Hence it is not the ability to view the source that is at question, but whether or not the source is put front and center as the medium itself. And in nearly all software systems it is *not*. (Perhaps special allowances would have to be paid for things like disassemblers, hex editors,

and software development kits.) Such realities are not ungermane to the fields of poetics or hermeneutics which often must deal with models of expression wherein the kernel of the work is relegated to the place of the ineffable, as in the “suggestion” or articulable “silence” of Symbolism or the masochistic disavowal of the cinema which can never truly recreate its subject, action through time and space, only depict it happening.

Such is the fundamental contradiction: what you see is not what you get. Software is the medium that is not a medium. Information interfaces are always “unworkable.” Code is never viewed as it is. Instead code must be compiled, interpreted, parsed, and otherwise driven into hiding by still larger globs of code. Hence the principle of *obfuscation*. But at the same time it is the exceedingly high degree of declarative reflexivity in software that allows it to operate so effectively as source or algorithmic essence – the stating of variables at the outset, the declarations of methods, all before the real “language” takes place – within a larger software environment always already predestined to parse and execute it. And hence the principle of *reflection*.

. . . That is Functional

Unworkable does not mean ineffectual. To describe a software interface as unworkable is not to describe it as inert or listless. So let us return to Chun’s claim that “software is a functional analog to ideology.” This indicates not only that software is an analog to ideology, but a much more fundamental claim, that software is *functional* in nature, therefore suggesting that ideology might be too. In other words, software is ideology turned machinic. I hinted at this above with reference to ideology’s “determinism narrative.” The discussion of the determinism narrative was meant to highlight the aspect of ideology that is oriented toward changing and inflecting the material world, the primary example of which would be the discussion of interpellation and subject formation in Althusser. In contrast, many have argued – Foucault famously – that it is really

the reverse: ideology is not a prime mover that casts subjects in its image, but rather real social and technical apparatuses discipline and inflect the material resources immanent to them, be they human bodies or otherwise. The pattern of proscriptive constraints may not be new, but what is crucial in software is the translation of ideological force into data structures and symbolic logic, a process no doubt coterminous with the evolution of language itself. Software is algorithmically affective in ways that ideology never was. This is best understood not as evidence of a schism between software and ideology, but as the very consummation of the deterministic, expedient narratives of both.

Media are the expression of this functional mandate. But how does it work? The following slogan helps explain the functional nature of software: *code is the only language that is executable*. Speech act theory has dealt with such questions already for some time. Particularly relevant to the present discussion is the concept of an illocutionary speech act, defined as a verbal expression that when uttered changes some state of affairs in the world. Here is Katherine Hayles on the illocutionary quality of all code:

Code has become arguably as important as natural language because it causes things to happen, which requires that it be executed as commands the machine can run.

Code that runs on a machine is performative in a much stronger sense than that attributed to language. When language is said to be performative, the kinds of actions it “performs” happen in the minds of humans, as when someone says “I declare this legislative session open” or “I pronounce you husband and wife.” Granted, these changes in minds can and do result in behavioral effects, but the performative force of language is nonetheless tied to the external changes through complex chains of mediation. By contrast, code running in a digital computer causes changes in machine behavior and, through networked ports and other interfaces, may initiate other changes, all implemented through transmission and execution of code.⁹

Illocutionary speech acts thus provide a basic structure for thinking about how code works. Yet not all agree on this point.

Opponents of the claim that all code is illocutionary point out that computer languages and natural languages are not different on the question of execution. They argue that illocutionary speech acts in natural languages require a general social understanding between groups of people in order for their performative quality to be effective – a pronouncement of marriage from the mouth of a priest creates a change in the world, but from an actor in a theater the same utterance exacts no such change – and likewise computer code requires a general infrastructure, the hardware of the computer, in order to carry out its “illocutionary” command. Yet I agree with Hayles: code is machinic first and linguistic second; an intersubjective infrastructure is not the same as a material one (even if making such a claim unfortunately splits these two symbolic systems into the “soft” natural languages versus the “hard” computer languages). To see code as subjectively performative or enunciative is to anthropomorphize it, to project it onto the rubric of psychology, rather than to understand it through its own logic of “calculation” or “command.” The material substrate of code, which must always exist as an amalgam of electrical signals and logical operations in silicon, however large or small, demonstrates that code exists first and foremost as commands issued to a machine. Code essentially has no other reason for being than instructing some machine in how to act. One cannot say the same for the natural languages. (Elsewhere Chun complicates this line of reasoning with her evocative argument that source code is only ever understood as source code *after the fact*.) Of course this is not to exclude the cultural or technical importance of any code that runs *counter* to the perceived mandates of machinic execution, such as the computer glitch or the software exploit, simply to highlight the fundamentally functional nature of all software (glitch and exploit included).

Machines and narratives are not alike however. A tension remains between software, which I suggest is fundamentally a machine, and ideology, which is generally understood as a narrative of some sort or another. Espen Aarseth’s heroic reworking of text and narrative into what he calls the “traversal functions” of electronic texts is indicative of how narrative

cannot exist in code as such, but must be simulated, either as a “narrative” flow function governing specific semantic elements, or as an “image” of elements in relation as in the case of an array or a database. Software is not primarily a verbal narrative or a visual image, even if certainly these latter forms can be remediated in software. The problem stems from the two basic understandings of software, one as computer language and the other as machine. As language, software is a symbolic system that can exist in different modes, often understood as linguistic or code “layers” (the crux of Kittler’s argument in “There is No Software”). As I argued above in the context of transcoding, one of the outcomes of this perspective is that each layer is technologically related, if not entirely equivalent, to all the other layers. However the linguistic layer model of software is most instructive for an altogether different claim it makes, this time about the fundamental incommensurability between any two points or thresholds on the continuum of layers, and therefore about the difficulty of achieving a collective or “whole” experience. For these are not simply inert technical translations back and forth; there is a privileged moment in which the written becomes the purely machinic and back again. The operating system may exist as an executable on disk, but it also exists phenomenologically as a metaphoric, cybernetic interface: the “desktop.” (Of course metaphor is entirely the wrong term for talking about figurative interactivity, but it will have to do for the moment; in the context of gaming I have proposed “allegorithm,” a portmanteau of “allegory” and “algorithm,” but it too seems slightly awkward for the present discussion.)

Underscoring its allegorical structure, Aarseth refers to this strange condition as the dual nature of the cybernetic sign. As allegory, it is best understood in a larger social context where the forced divorcement between the poetic and the functional, or the private and the public, or process and stasis, is a projection of the agonizing scars of fragmentation and atomization in all layers of social life. The dialectical movement between fluidity and fixity, seen in the internal workings of software where states and state changes carry the day, is precisely the

same political problem posed by ideology (the narratives of failure and determinism summarized at the start of the chapter). Software might not be narrative in the strict sense of the word, but it still might have a beginning, middle, and end – to paraphrase Aristotle – even if those narrative moments are recast as mere variables inside the larger world of the software simulation. Thus too might ideology be recast in digital format.

Riven to the core, software is split between language and machine, even if the machinic is primary. And, more importantly, there is a process of mystification or distancing at work which ensures that the linguistic and the machinic are most definitely not the same thing. Indeed, part of the ideological import of software is the creation and maintenance of such a distinction. This leads to my second general observation which has to do with the depth model of representation (introduced above via Chun's "persistence of visual knowledge"): *software is both scriptural and executable*. As already discussed these two modes are often splayed out into a hierarchical model of two or more layers of code: source code is "prior to" a runtime executable; machine languages are "underneath" programming languages; software applications "drive" the user experience, and so on. The relevant section from Aarseth is worth quoting at length due to its clarity and depth.

The dual nature of the cybernetic sign processes can be described as follows: while some signification systems, such as painted pictures and printed books, exist on only one material level (i.e., the level of paint and canvas, or of ink and paper), others exist on two or more levels, as a book being read aloud (ink-paper *and* voice-soundwaves) or a moving picture being projected (the film strip *and* the image on the silver screen). In these latter cases, the relationship between the two levels may be termed *trivial*, as the transformation from one level to the other (what we might call the secondary sign production) will always be, if not deterministic, then at least dominated by the material authority of the first level. In the cybernetic sign transformation, however, the relationship might be termed *arbitrary*, because the internal, coded level can only be fully experienced by way of the external, expressive level. (When inactive, the program and data of the internal level can of

course be studied and described as objects in their own right but not as ontological equivalents of their representations at the external level.) Furthermore, what goes on at the external level can be fully understood only in light of the internal. Both are equally intrinsic, as opposed to the extrinsic status of a performance of a play vis-à-vis the play script. To complicate matters, two different expression objects might result from the same code object under virtually identical circumstances. The possibilities for unique or unintentional sign behavior are endless, which must be bad news for the typologists.¹⁰

Interfaces are thus manifest (as screens or keyboards), but also latent within software as the mediation between internal and external levels, as Aarseth terms them. The difference between “trivial” and “arbitrary” is essentially that between analog representation and digital representation. The analog is only deemed “trivial” because of the perceived obviousness of mimetic congruence using a continuously variable material value (for example, the curvilinear representation of a sound wave). Likewise the digital is deemed “arbitrary” because of the seeming disconnect between an empirical referent and the mathematical approximation of its form using discrete quanta. However both modes, the “arbitrary” included, are governed by what was referred to previously as “technical transcoding without figuration.” What makes Aarseth’s claim provocative, and indeed what I hope to rearticulate in this chapter, is that the technical, or “arbitrary,” transcoding of symbolic systems is in no way whatsoever a theory of inert material determinism. In fact it is the exact opposite: the fact that abstraction and figuration *do* exist in software (the interface metaphor of the “desktop” as functional emanation of source code, or any number of examples cited previously) demonstrates the fundamental indeterminacy of a technological apparatus that is, admittedly, grounded in rote, deterministic mathematical language. It is representation in form, but not in deed, and this is the paradox. It is representation as mathematical recoding, not as any socially or culturally significant process of figuration, yet at the end of the day what emerges is exactly that.

This is what Aarseth means when he says that “the possibilities for unique or unintentional sign behavior are endless.” Let me underscore that this needs to be understood not in the “soft” anthropocentric sense of the varying interpretive and cognitive intangibles brought to the table by human agents, but in the “hard” sense of complex material systems and the innumerable combinations that emerge from them. The fetish in Marx is never blamed on the shortcomings of the human mind, even if it follows a logic of misrecognition. We should not do the same with software.

Given that the book is well underway, it is time to pause and reflect again on the question of method first broached in the preface. This book tries to follow a very specific path, one in which the political interpretation of cultural and technical objects is put forth not as one possible style of reading to be swapped in and out according to one’s methodological preference, but instead that the political interpretation of cultural and technical objects is essentially synonymous with interpretation itself such that to do one is necessarily to do the other, and likewise to ignore the former is to perform the latter poorly. Software, in other words, *asks a question to which the political interpretation is the only coherent answer.*

Note that this approach is different from those who seek to unmask how this or that piece of software might be a bearer of some political worldview. This is not a theory of *ideologies*, each paired up with an appropriately insightful critique crafted to debunk it. The task here is not to claim that software has a “meaning,” political or otherwise, that can be revealed through a convenient methodological scaffolding called the political interpretation. Quite the opposite is the case. A certain networked relation is at play: software, the social, and the act of interpretation combine in “an intense mimetic thicket” and it is this thicket that, in its own elaboration, can be called the political. Chun’s claim made at the outset that “software is a functional analog to ideology” contains all of the many strands of this emergent structure: (1) software and ideology are related in a fundamental way; (2) yet it is a relationship of figuration

in which the complexities and contradictions of ideology, which itself contains both utopian and repressive instincts, are modeled and simulated out of the formal structure of software itself; (3) further, software is functional and thereby exacerbates and ridicules the tension within itself between the narrative and machinic layers – the strictly functional transcodings of software, via a compiler or a script interpreter for example, fly in the face of the common sense fact that software has both an executable layer, which should obey the rules of a purely functional aspect of the code (similar to what Genette calls the “paratextual” in literature), and a scriptural layer, which would obey the rules of semantics and subjective expression (in Deleuze, remember, it is the *nonsubjective* that is the machinic).

In fact all this comes under a more common name, *allegory*. The point is not simply that software is functional, but that software’s mock resolution of the tension between the machinic and the narrative, the functional and the disciplinary, the fluid and the fixed, the digital and the analog, is an allegorical figure for the way in which these same political and social realities are “resolved” today: not through oppression or false consciousness, as in the orthodox ideological critique, but through the ruthless rule of code, which proposes that the analog should live on to the end, only to show that the analog never existed in the first place. And as writers like Jameson and Northrop Frye have pointed out, the act of interpretation is but another moment of allegorical structuring, as parallel or “analog” discourses are extracted or, if you like, expressed through and within media technologies (or, formerly, within texts). From Plato onwards, such is the logic of ideology. To claim that ideology exists is to claim that political apathy and machinic canalization are present here alongside the very possibility of their transcendence – otherwise it would not be ideology, but something like psychosis. So it is really desire that is the stuff of ideology. It is a desire not for the absence of ideology in something like the end of history, but for the very presence of ideology as a reminder for how the sacred “end” must always already be contained in the profane present. The logic shines through quite elegantly in the words of Ernst

Bloch, writing from an earlier moment in the machine age. "Someone once said that people are in Heaven and don't know it; Heaven certainly still seems somewhat unclear. Leave everything from his statement but the *will* that it be true – then he was right."