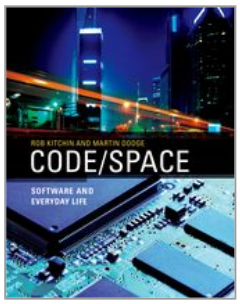


Code/Space: Software and Everyday Life
Rob Kitchin and Martin Dodge

Print publication date: 2011
Print ISBN-13: 9780262042482
Published to MIT Press Scholarship Online: August 2013
DOI: 10.7551/mitpress/9780262042482.001.0001

The Nature of Software

Rob Kitchin
Martin Dodge

DOI:10.7551/mitpress/9780262042482.003.0002

[–] Abstract and Keywords

This chapter discusses the conceptualization of the varied nature of software. It emphasizes that programming, which is done by software programmers, is performative and negotiated, and that code which can be recorded to media is mutable. The chapter also discusses code as a product and as a process, and defines several code-related terms, which include commentary code, prescriptive code, code objects, and critical code. It discusses various approaches and challenges of creating new code, which is a complex and difficult task, especially when attempting to address new issues and problems. The chapter concludes that software augments, supplements, mediates, and regulates the lives of people and has the ability to transform the world in relation to its own systems of thought because it possesses high technicity.

Keywords: software programming, commentary code, prescriptive code, code objects, critical code, technicity

> Code, the language of our time.
>
> Code = Law
>
> Code = Art
>
> Code = Life
>
> —Gerfried Stocker and Christine Schöpf

> [Software] is philosophical in the way it represents the world, in the way it creates and manipulates models of reality, of people, of action. Every piece of software reflects an uncountable number of philosophical commitments and perspectives without which it could never be created.
>
> —Paul Dourish
>
> The art of creating software continues to be a dark mystery, even to the experts. Never in history have we depended so completely on a product that so few know how to make well.
>
> —Scott Rosenberg

In this chapter, we explore the variegated nature of software. In particular, we argue that a comprehension of software must appreciate two aspects of code; first, that code is a product of the world and second, that code does work in the world. Software as both product and process, we argue, needs to be understood within a framework that recognizes the contingent, relational, and situated nature of its development and use. Software does not arise from nowhere; code emerges as the product of many minds working within diverse contexts. As Mackenzie (2003, 3) notes, software is created through “complex interactions involving the commodity production, organizational life, technoscientific knowledges and enterprises, the organization of work, manifold identities and geo-political-technological zones of contact.” Just as software comes from diverse threads, software’s effect in the world is not deterministic or universal. Rather, software as an actant, like people as actors, functions within diversely produced social, cultural, economic, and political contexts. The effects of software unfold **(p.24)** in multiple ways in many milieus. Often, this unfolding action is messy, imperfect, and always near the edge of breakdown (as is readily apparent when maintaining a working software setup on desktop PC). Surrounding and coalescing around software are discursive and material assemblages of knowledge (flow diagrams, Gantt charts, experience, manuals, magazines, mailing lists, blogs and forums, scribbled sticky notes), forms of governmentalities (capta standards, file formats, interfaces, conventional statutes, protocols, intellectual property regimes such as copyrights, trademarks, patents), practices (ways of doing, coding cultures, hacker ethos, norms of sharing and stealing code, user upgrading, and patching), subjectivities (relating to coders, sellers, marketers, users), materialities (computer hardware, disks, CDs, desks, offices), organizations (corporations, consultants, manufacturers, retailers, government agencies, universities and conferences, clubs and societies) and the wider marketplace (for code and coders).

Code lacks materiality in itself. It exists in the way that speech or music exist. All three have diverse effects and can be represented and recorded to a media (for example, written as text on paper). Yet software when merely written as lines of code loses its essential essence—its executability. Layers of software are executed on various forms of hardware—CPUs, motherboards, disk drives, network interfaces, graphics and sound cards, display screens, scanners and other devices, network infrastructures, printers and other peripherals, and so on—using various algorithms, languages, capta rules, and communication protocols, thus enabling them to coalesce and function in diverse ways in conjunction with people and things by interfacing the virtual (the world as 0s and 1s) with the material. At the heart of this assemblage is code—the executable pattern of instructions.

Code

> What software does and how it performs, circulates, changes and solidifies cannot be understood apart from its constitutions through and through as code…. Code cuts across every aspect of what software is and what software does.
>
> —Adrian Mackenzie

Code at its most simplistic definition is a set of unambiguous instructions for the processing of elements of capta in computer memory. Computer code (henceforth code) is essential for the operation of any object or system that utilizes microprocessors. It is constructed through programming—the art and science of putting together algorithms and read/write instructions that process capta (whether that is variables held at specific addresses in memory space, human keystrokes or mouse movements, disk files, or network fields) and output an appropriate response (a series of letters appearing on a screen, the field of view changing in a game, a credit card being verified, **(p.25)** an airplane ticket being booked, or MP3 files being transferred from a compact disk, decoded, and played). Coded instructions transduce input; that is, the code changes the input from one state to another, and as a consequence the code performs work. As Berry (2008) details, code "does *something* to *something* … it performs functions and processing" (emphasis in the original). The skill of a programmer is to construct a set of coded instructions that a microprocessor can unambiguously interpret and perform in ongoing flows of operations.

There is a large variation in how programming is discursively produced from assembly languages (producing machine code) to scripting and procedural languages (producing source code) that has to be compiled (into executable code that the machine can understand). As a consequence, the nature of programming varies depending on how structured a language is, the scope and scale of action available to the programmer, and the extent to which the language is talking directly (instructing) to the hardware rather than through an interpreter or compiler. All programming languages have formal rules of syntax, grammar, punctuation, and structure. In the case of scripting and procedural languages, the coding is less abstract than that written in assembly languages, and has characteristics more akin to natural languages. In this case, programmers use formalized syntax, usually based around natural language words and abbreviations, along with symbols and punctuation, to construct structured programs made up of statements, loops, and conditional operations (Berry 2004). For example, below is a piece of code that calculates whether a point is inside a polygon (a common evaluative procedure in a geographic information system).

```
procedure point_in_polygon;
var
i, j : integer;
slope, inter, yi : real;
begin
j := 1;
for i := 2 to npts do begin
if (data [i].x 〈〉 data [i + 1].x) then
if ( ( (data [i].x - data [1].x) ) * ( (data [1].x - data [i + 1].x) ) 〉 = 0)
then
if ( (data [i + 1].x 〈〉 data [1].x) or (data [i].x 〉 = data [1].x) )
then
begin
slope := (data [i + 1].y - data [i].y) / (data [i + 1].x - data [i].x);
inter := data [i].y - slope * data [i].x;
yi := Inter + slope * data [1].x; (p.26)
if (yi 〉 data [1].y) then
j := j * ( - 1)
end;
end;
if j = - 1 then
writeln (‘point in polygon’)
else
writeln (‘point not in polygon’);
end;
```

As Brown (2006) notes, programming languages "do *double duty* in that they work as an understandable notation for humans, but also as a mechanically executable representation suitable for computers.... Computer code *has to be* automatically translatable to a form which can be executed by a machine … [they] thus sit in an unusual and interesting place—designed for human reading and use, but bound by what is computationally possible" (emphasis in the original). And just as there are different languages, there are different kinds of programming. Programming extends from the initial production of code, to refactoring (rewriting a piece of code to make it briefer and clearer without changing what it does), to editing and updating (tweaking what the code does), to integrating (taking a piece of code that works by itself and connecting it to other code), testing, and debugging, to wholesale rewriting. The skill to model complex problems in code that is effective, efficient, and above all, elegant, is seen as a marker of genuine programming craft. Some have argued that this acumen is akin to being artistically gifted, such as composing beautiful verse or sculpting apollonian forms: "The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination" (Brooks 1995, 7). Crafting code can itself be a deeply creative act (see chapter 6), although one should be wary of glorifying what is for many programmers a daily grind of just getting things written and working.

Regardless of the nature of programming, the code created is the manifestation of a system of thought—an expression of how the world can be captured, represented, processed, and modeled computationally with the outcome subsequently doing work in the world. Programming then fundamentally seeks to capture and enact knowledge about the world—practices, ideas, measurements, locations, equations, and images—in order to augment, mediate, and regulate people's lives. Software has, at a fundamental level, an ontological power, it is able to realize whole systems of thought (algorithms and capta) with respect to specific domains. For example, consider the influence of formalizing and coding how money is represented and transacted and thus how the banking system is organized and works. Many other examples come to mind, such as how a game should be played, how a car operates and should be driven, how a document is to be written, or how a presentation is to be given (see Fuller's [2003] analysis **(p.27)** of the ontological capacity of Microsoft Word and Tufte's [2003] critique of Microsoft PowerPoint) and so on. It does this by formalizing a system into a set of interlinked operations through the creation of itineraries expressed as algorithms, "recipes or sets of steps expressed in flowcharts, code or pseudocode" (Mackenzie 2006, 43). The source code above is an example of a coded instruction expressed as an algorithm to solve a specific problem (calculating whether a point is inside a polygon).

Software thus abstracts the world into defined, stable ontologies of capta and sequences of commands that define relations between capta and details how that capta should be processed. The various structures in which capta is stored ("lists, tuples, queues, sequences, dictionaries, hashtables") can then be worked upon by various algorithms designed to sort, search, swap, increment, group, and match (Mackenzie 2006, 6). Agency is held in these simple algorithms in the sense that they determine, for themselves, what operations do and do not occur. Code performs a set of operations on capta to enact an event, an output of some kind (a value is displayed on screen, the antilock brakes are applied following the driver pressing the pedal, a selected song is played through speakers, writing is saved into a word processor document). Even in everyday software applications, such as a word processor or web browsers, code enacts

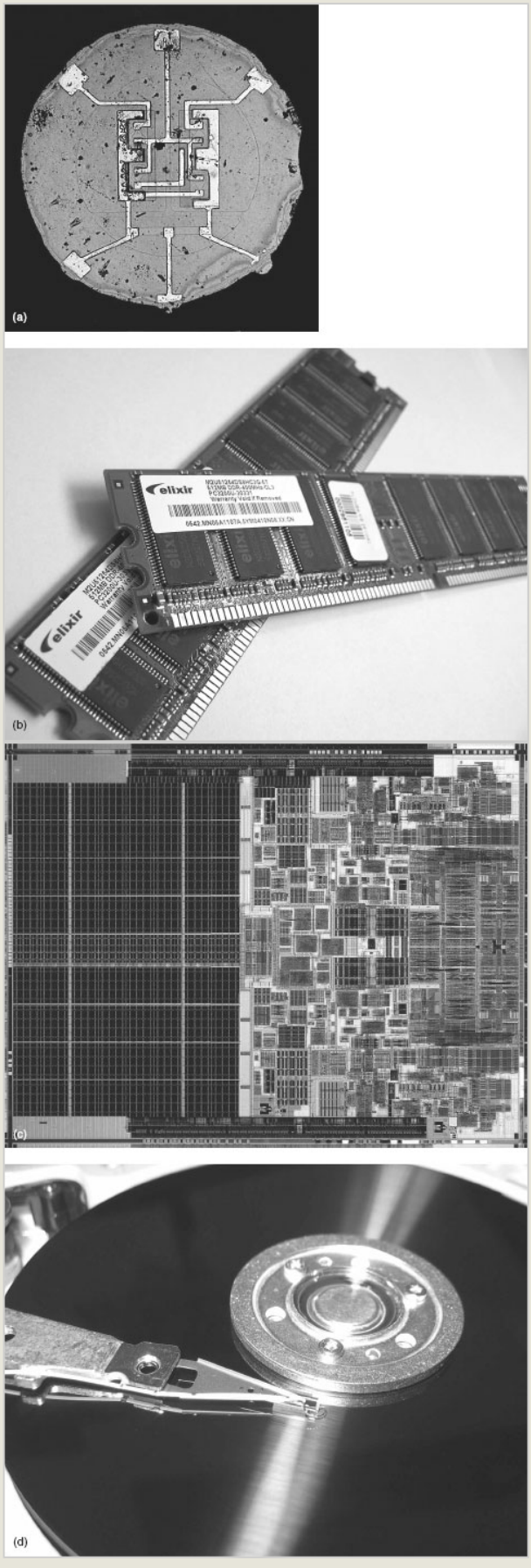
millions of algorithmic operations to derive an outcome at a scale of operation so small and fast as to be beyond direct human sensing (see figure 2.1). Indeed, we only sense the nature of these processes when the response of a software application slows to such an extent it makes us wait. (Research in human-computer interactions, HCI, demonstrates that the necessary response time for interactive use of software to be about 0.1 sec. for people to feel they are in charge. Delays of 1 sec. in response are noticeable but tolerable; and longer delays will lead to user frustration and distraction from a task; Miller 1968).

As Mackenzie (2006, 43) notes “algorithms carry, fold, frame and redistribute actions into different environments.” Fuller (2003, 19) thus argues that software can be understood as “a form of digital subjectivity—that software constructs sensoriums, that each piece of software constructs ways of seeing, knowing, and doing in the world that at once contain a model of that part of the world it ostensibly pertains to and that also shape it every time it is used.” This digital subjectivity is “an ensemble of pre-formatted, automated, contingent, and ‘live’ action, schemas, and decisions performed by software, languages and designers…. [It] is also productive of further sequences of seeing, knowing and doing” (Fuller 2003, 54). In this sense, “code is saturated with, and indivisible from social phenomena,” in the sense that how we come to know the world is always social (Rooksby and Martin 2006, 1).

This relationship between code algorithms, capta structures, and the world is well illustrated with respect to weather prediction and global climate change modeling (see Washington, Buja, and Craig 2009 for an overview). Gramelsberger (2006) details how scientific theories about weather systems and empirical observations from across **(p.28)**

(p.29) (p.30) the world are translated into a formal model, which is then translated into mathematical formulae, which is then translated into executable code, which evokes a kind of imagined narrative about future climates. Here, knowledge about the world is translated and formalized into capta structures and algorithms that are then converted into sets of computational instructions that when applied to climate measurements express a particular story. Gramelsberger expresses this as “Theory = Mathematics = Code = Story.” Our understanding of weather forecasting and climate models are almost entirely driven by these computational models, which have been refined over time in a recursive fashion in response to how these models have performed, and which are used to theorize, simulate and predict weather patterns (indeed, meteorologists and geophysists are major users of supercomputers, TOP500 2009). In turn, the models underpin policy arguments concerning climate change and have real effects concerning individual and institutional responses to measured and predicted change. The models are coded theory and they create an experimental system for performing theory (Gramelsberger 2006); or, put another way, the models analyze the world and the world responds to the models.

Such simulation models of complex physical systems are now common in the “hard” sciences such as earth sciences, physics, astronomy, and bioinformatics, and may become more significant in social sciences in the coming years (Lane et al. 2006; Lazer et al. 2009). A key element of the success of such software models is their ability to generate (spatial) capta that can be visualized to create compelling inscriptions; figure 2.2 shows an example from climate change modeling. For businesses managing



complex distributed operations, with multiple risks and dynamic flows, software models that can anticipate problems before they occur are also becoming important (see Budd and Adey 2009 for a discussion of software models that keep air travel moving). In the realm of international finance, predictive software models of market trading conditions are responsible for millions of automatic transactions worth billions of dollars and the decisions that algorithms autonomously undertake affect the fluctuation in prices (see D. MacKenzie's 2006 work on how complex mathematical equations in financial models translate into operative trading software models that then drive the markets that they purportedly represent). Indeed, the ease with which software models enabled mortgage risks to be manipulated has been blamed for contributing to the recent "credit crunch" (Osinski 2009).

Figure 2.1 Computation microspaces where software works and capta dwells. (*a*) Silicon logic gate. (*b*) Memory cards. (*c*) Circuit board. (d) Hard drive.

(*Source* : www.technologyreview.com/article/21886/).

(Source : http://images.suite101.com/538579_com_ram.jpg).

(Source : www.technologyreview.com/article/21886/).

(Source : http://commons.wikimedia.org/wiki/File:Hard_disk.jpg).

David Berry (2008) suggests that the "properties of code can be understood as operating according to a grammar reflected in its materialisation and operation," detailing seven ideal types through which code is manifested. The *digital data structure* is a static form of data representation, wherein data are held digitally as binary 0s and 1s. The *digital stream* is the flow of digital data structures through a program and across media (for example, from CD to hard disk or across a network). *Delegated code* is human-readable source code that is editable. *Commentary code* is the textual area in **(p.31)**

(p.32) delegated code where programmers detail authorship, describe what a piece of code does, and document changes. *Prescriptive code* is compiled delegate code that can be read and processed by microprocessors to do work on the digital data structure. *Code objects* are the everyday objects that are represented within the delegate code and which the prescriptive code seeks to work on or for (e.g., a car, a washing machine, a PDA). *Critical code* is code written to reverse engineer prescriptive code into delegate code, or to read/hack digital streams and digital data structures. In combination, these ideal types illustrate how software development consists of the production and phasing of a set of codings. What Berry's (2008) grammar of code also makes clear is that code is a product—it is brought into the world by the labor and skills of programmers.

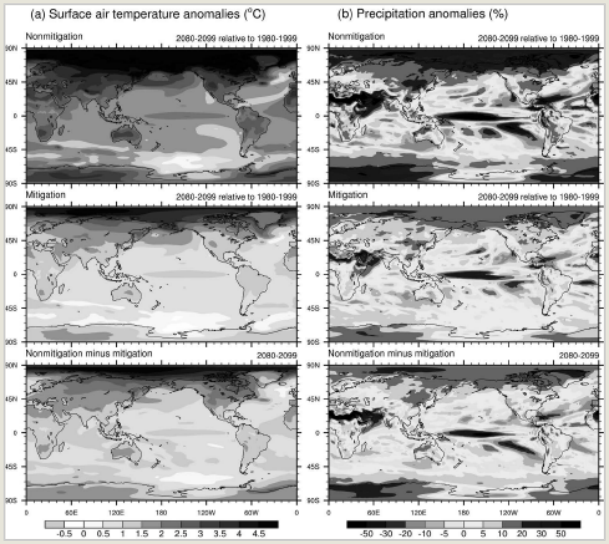


Figure 2.2 Comparative mapping of predictions of (*a*) surface temperature and (*b*) precipitation at the end of the twenty-first century from global climate change model made possible by software.

(*Source* : Washington et al. 2009, 3.)

Code as Product

> If people really knew how software got written, I' m not sure they'd give their money to a bank or get on an airplane ever again
>
> —Ellen Ullmann
>
> We now depend on unfathomably complex software to run our world. Why, after a half century of study and practice, is it still so difficult to produce computer software on time and under budget? To make it reliable and secure? To shape it so that people can learn it easily, and to render it flexible so people can bend it to their needs? … [I]s there something at the root of what software is, its abstractness and intricateness and malleability, that dooms its makers to a world of intractable delays and ineradicable bugs —some instability or fickleness that will always let us down?
>
> —Scott Rosenberg

In reference to the last quote above, Rosenberg (2007) seeks to answer his questions through an in-depth ethnographic study undertaken over a three-year period of one company's attempt to produce a new application. What his study highlights is that producing software is a complex and contingent process. It has these qualities because code is produced by people who vary in their abilities and worldviews and are situated in social, political, and economic contexts. While the activity of programming is often undertaken individually—a single person sitting in front of a computer writing the code—this occurs within a collaborative framework, with individuals

performing as part of a team working toward a larger, common goal. Team members interact when needed to tackle problems, find solutions, react to feedback, discuss progress, and define next steps and roles. Often several teams will work on different aspects of the same program that are stitched together later. These teams produce programs often consisting of thousands of routines, interlinked in complex ways, that work together **(p.33)** to undertake particular functions. Teams may well be using different coding languages or drawing on existing codebases (either within a company from previous projects or open source), have different visions about what they are trying to achieve, and have different skill levels to tackle the job at hand. Indeed, it is fair to say that individual programmers diverge in coding abilities, experience, motivation, and productivity, which leads to a varying quality and quantity of work. The result is often a project so complex that no single programmer can know everything about it, particularly at a fine scale. Indeed, the sheer scale of labor expended to create code can be immense. Starting from humble origins in the late 1940s, today's software applications often consist of millions of lines of code. For example, it is estimated that the first version of Microsoft Word, released in 1983, consisted of around 27,000 lines of code but had grown to about 2 million by 1995 as more functions and tools were added (Nathan Myhrvold cited in Brand 1995). Operating systems, the guts of mass market computing, consist of enormous sets of code (table 2.1), which in part is responsible for their unreliability and also their susceptibility to malicious hacking and viruses.

Even if a programmer is working on his or her own, the program is implicitly the result of a collective endeavor—the programmer uses a formalized coding language, proprietary coding packages (and their inherent facilities and defaults), and employs established disciplinary regimes of programming—ways of knowing and doing regarding coding practices and styles, annotation, elegance, robustness, extendibility, and so on (as formalized through manuals, instruction, conventions, peer evaluation, and industry standards). Coding then is always a collaborative manufacture with code being a social object among programmers (Martin and Rooksby 2006). Knowledge of the code and its history (its iterations, its idiosyncrasies, its weaknesses, and its inconsistencies) is shared and distributed among the group, although it is possible for other coders to work through, understand, and modify the code by reading and playing with it (Martin and Rooksby 2006).

Table 2.1 Growth in size of Microsoft operating system that is used by a large proportion of the world's PCs.

Year	Operating system	SLOC (Million) [source lines of code]
1993	Windows NT 3.1	4–5
1994	Windows NT 3.5	7–8
1996	Windows NT 4.0	11–12
2000	Windows 2000	More than 29
2001	Windows XP	40
2003	Windows Server 2003	50

***Source* : www.knowing.net**

(p.34)

Ullman's (1997, 14–15) self-ethnography, *Close to the Machine* , illustrates this collective manufacture in detail, highlighting how groups of programmers work together in various (dys)functional ways to produce code. Sometimes this is as individuals working mostly alone but within teams, and sometimes as a group. For example, in relation to the latter, she writes:

> We have entered the code zone. Here thought is telegraphic and exquisitely precise. I feel no need to slow myself down. On the contrary, the faster the better. Joel runs off a stream of detail, and halfway through the sentence, Mark, the database programmer, completes the thought. I mention a screen element, and Danny, who programs the desktop software, thinks of two elements I' ve forgotten. Mark will later say all bugs are Danny' s fault, but, for now, they work together like cheerful little parallel-processing machines, breaking the problem into pieces that they attack simultaneously.
>
> "Should we modify the call to AddUser—"
> "—to check for UserType—"
> "Or should we add a new procedure call—"
> "—something like ModifyPermissions."
> "But won't that add a new set of data elements that repeat—"

Programming as a creative endeavor means there is inherent unpredictability in its production. There is no single solution to most coding problems and the style of the solution is often seen as important. Many programmers are said to exhibit a hacker ethic, a desire to craft inventive, elegant solutions, deploying algorithms and capta structures that one might view as having aspect beauty. The accounts of Ullman and Rosenberg also reveal the negotiated and contingent processes through which code unfolds; how programming labor is performative in nature (Mackenzie 2006). Code is developed through collective cycles of editing, compiling, and testing, undertaken within diverse, historically-framed, social contexts. It is citational, we would argue, consisting of embedded, embodied, and discursive practices. This type of practice "that echoes prior actions, and accumulates the force of authority through the repetition or citation of a prior and authorative set of practices … [and] it draws on and covers over the constitutive conventions by which it is mobilized" (Butler 1990, 51). Software products are inherently partial solutions and imperfect; their code is mutable and contingent—scripted, rewritten, updated, patched, refined; "code … slips into tangles of competing idioms, practices, techniques and patterns of circulation … there is no'program' as such, only programmings or codings" (Mackenzie 2006, 5).

Given its citational qualities, once a design approach and programming language are chosen and a certain amount of progress made, a contingent pathway is created that is difficult to diverge from to any great degree without going back to the beginning and starting again. This is especially true because software consists of layers of abstraction. Significantly altering one layer has knock-on consequences for all component **(p.35)** layers. And yet as Rosenberg (2007) notes, the choice of programming language often comes down to the arbitrary or ineffable (matter of taste or gut sense) or simply previous experience, rather than being a rational choice informed by reflection and research. (There are quasi tribal loyalties exhibited by some programmers to their favored languages and a dismissal of those who write in so-called inferior

ones.) As such, arbitrary choices and contingent progress push projects more or less in a particular direction that then limit future decisions and outcomes.

Adopting the most appropriate approach given the present computational landscape is not straightforward. As Mackenzie (2006) notes, because of ongoing innovations in software languages, code libraries, and development environments, programmers are always trying to keep abreast of the latest developments. For example, Ullman (1997, 100–101) details that in a twenty-year period she had taught herself, “six higher-level programming languages, three assemblers, two data-retrieval languages, eight job-processing languages, seventeen scripting languages, ten types of macro, two object-definition languages, sixty eight program-library interfaces, five varieties of networks, and eight operating environments.” This is an enormous sum of learning and adaptation, yet paradoxically, as Joel Spolsky (cited in Rosenberg 2007, 274) states, “most programmers don’t read much about their discipline. That leaves them trapped in infinite loops of self-ignorance.” Similarly, Ullman (1997, 110) argues that “the corollary of constant change is ignorance. This is not often talked about: we computer experts barely know what we are doing. We’re good at fussing and figuring out. We function in a sea of unknowns.… Over the years, the horrifying knowledge of ignorant expertise became normal, a kind of background level of anxiety.” In other words, programming takes place in an environment that it is changing so quickly that it is often difficult to keep up with new developments, and with such diversity, programmers can differ markedly in their ability to write good code and in how they think a system should be coded. Ignorance is often compounded because programmers can adopt a silo mentality and start a project afresh, producing new code rather than using the vast amounts of code that has already been produced (for example, Source-forge is an open-source warehouse of code for over 230,000 projects as of February 2009; www.sourceforge.net). In part this is due to the hacker ethos of believing that it is possible and desirable to start afresh in order to produce a new and better solution to a given problem.

The coding itself can be a complex and difficult task, especially when trying to address new problems for which solutions have not been established. As a result, there are various competing schools of thought with regard to how software development should take place, much of it focusing on project management rather than the practice of writing code. For example, the structured programming approach aims to limit the creation of spaghetti code by composing the program of subunits with single exit and entry points. Disciplined project management focuses on advance planning and **(p.36)** making sure that a clear and defined route from start to end is plotted before coding takes place. The capability maturity model seeks to move coding organizations up a ladder of management practices. Pattern models divide up projects into orderly and iterative sequences; while rapid application development relies on quick prototyping and shorter iterative phases to scope out ways forward. Another approach, agile software development, that also prioritizes speed, uses experimentation and innovation to find the best ways forward and to be responsive to change (Rosenberg 2007).

However, even when a particular approach has been adopted, the challenges of creating new code and entwining different elements and algorithms together can produce problems that are intellectually demanding and difficult to solve. Abstracting the world and working on and communicating those abstractions (from programmer to machine, programmer to programmer,

and program to user) in the desired way can be vexing tasks, often without correct solutions. Consequently, software is inherently partial and provisional, and often seen to be buggy—containing code that does not work in all cases, or only works periodically, or stops working if new code is added or modified. Rosenberg (2007) describes several of what his participants called “black holes”—a coding problem of “indeterminate and perhaps unknowable dimensions” that could take weeks or months to address and sometimes led to whole areas of an intended application to be abandoned. So-called snakes are a significant class of black holes where there is no consensus within a team about how they could and should be tackled, producing internal conflict and disharmony among members that jeopardizes progress and sometimes entire projects. As Mackenzie (2006, 5) notes, code “is permeated by all the forms of contestation, feeling, identification, intensity, contex-tualization and decontextualizations, signification, power relations, imaginings and embodiments that compromise any cultural object.” Indeed, the project Rosenberg (2007) tracked had several high profile programmers working for it, and yet their collective skills and knowledge were severely tested by the challenges they faced. At the time he completed his book, they still only had a limited, beta application that was missing many of its desired components.

Unlike other kinds of product development, adding additional coders to a project to try and resolve problems, somewhat paradoxically, rarely helps. Indeed, a well established maxim in programming—Brook’s Law—states that “adding manpower to a late software project makes it later” (cited in Rosenberg 2007, 17); new workers absorb veterans’time, have to learn the complexities and history of the project, bring additional ideas that enhance the potential for further disagreement, and add confusion through a lack of familiarity with key issues. One solution has been to release software through a series of evolving versions and to periodically release updates and patches to fix problems discovered in an existing version. This has also led to software enjoying some unusual legal qualities, with its liabilities for failure being limited to a **(p.37)** significant degree. In other words, purchasers willingly accept an imperfect product while abrogating the supplier from responsibility for any damage caused through its use. These imperfections in terms of bugs, glitches, and crashes are at once notorious and yet also largely accepted as a routine dimension of computation. The imperfections also provide instrumental evidence that software is difficult to produce (Charette 2005). These imperfections matter because as well as causing particular problems, they also facilitate new kinds of criminal activity, along with petty digital vandalism. Illegal hacking, software viruses, and network attacks have become an ever present threat in recent years. The scale of corrupted code and deliberate virus infections is hard to gauge with any reliability, but Markoff (2007) reports that some 11 percent of computers on the Internet are infected. Software to secure other software is now itself a significant business opportunity!

Bugs, black holes, and snakes in code have pragmatic implications and can lead to serious slippage in project delivery dates and often a redefinition of intended goals. Of course, slippage can also occur because the client alters the project specification. Such changes highlight that the production of code does not take place in a vacuum. It is embedded within workplace or hacker cultures, personal interactions and office politics, relationships with customers/users, and the wider political and cultural economy. Ullman (1997) and Rosenberg’s (2007) ethnographies graphically reveal the different group dynamics within and between teams in a company, and how the development of code takes place in an environment that has frequent

staff changes, downsizing and expansion, mergers and takeovers, reprioritization of policy and development, and investor confidence (also explored in Coupland's novels, *Microserfs* (1995) and *JPod* (2006) on the fictional lives of programmers and software designers). These negotiations and contestations between individuals and teams are set in the wider political economy of finance and capital investment, market conditions, political/ideological decisions, and also the role of governments and the military-industrial complex in promoting the knowledge society, innovation culture, and underwriting significant amounts of development and training. For example, the ideology underlying the production and distribution of software varies between projects, as a comparison in the ethos, discourses, and practices of propriety software (executable code sold under license; source code not distributed), free software (code is a collective good; source code distributed), and open-source software (code is property of an individual who has the right to control and develop it; source code distributed) demonstrate (Berry 2004).

Software then is not an immaterial, stable, and neutral product. Rather, it is a complex, multifaceted, mutable set of relations created through diverse sets of discursive, economic, and material practices. The result of all of this contingency is that software development has high failure rates—either projects are never completed, do **(p.38)** not do what they were intended to do, or they are excessively error prone or unreliable in operation (hence the wide prevalence of beta releases, patches, and updates). A comparison of U.S. Federal Government software spending over a fifteen-year period demonstrated no improvement in the reliability or effectiveness of code delivered, "In 1979, 47 percent of federal software was delivered but never used due to performance problems, 29 percent was paid for but never delivered, 19 percent were used but extensively reworked, 3 percent was used after changes, and only a miniscule 2 percent was used as delivered. In 1995 in the same categories, the statistics were 46 percent, 29 percent, 20 percent, 3 percent and 2 percent" (Lillington 2008, 6). The Standish Group surveyed 365 information technology managers and reported in 1995 that only 16 percent of their projects were successful (delivered on time and budget and to the specification requested), 31 percent were canceled, and 53 percent were considered "project challenged" (over budget, late, failed to deliver on specification). By 2004, the figures were 29 percent successful, 18 percent canceled, and 53 percent "project challenged" (Rosenberg 2007). In both cases, over two-thirds of projects failed to deliver on their initial objectives.

Given those statistics, it is no surprise that the software landscape is littered with high profile, massively expensive, failed projects and there is even literature detailing these disasters (see, for example, Britcher 1999; Glass 1998; Yourdon 1997). Brief examples illustrate this point. The U.S. Internal Revenue Service's overhaul of its systems was canceled in 1995 after ten years at a cost of $2 billion. The U.S. Federal Aviation Administration canceled a new air traffic control system in 1994 after 13 years and billions of dollars (at its peak the project was costing $1 million a day). In 1996, a $500 million European Space Agency rocket exploded 40 seconds after takeoff because of a bug in the guidance system software (Rosenberg 2007). In 2004, UK supermarket chain Sainsbury wrote off a store inventory system at a cost of $526 million (Lillington 2008). A study by the U.S. National Institute of Standards and Technology published in 2002 details that software errors cost the U.S. economy about $59.5 billion annually in 2001

(Rosenberg 2007). It should come as no surprise that the failure of software systems is now routinely blamed for operating problems within a government agency or company.

It seems then that workable software that is effective and usable is all too often the exception rather than the rule. Even successful software has bugs and loopholes that mean its use is always porous and open to rupture in some unanticipated way. Setting and maintaining a home PC in working order, for example, takes continual digital housework (see chapter 8). Code is contingent and unstable—constantly on the verge of collapse as it deals with new data, scenarios, bugs, viruses, communication and hardware platforms and configurations, and users intent on pushing it to its limits. As a result, software is always open to new possibilities and gives rise to diverse realities. **(p.39)**

Code as Process

> Software forges modalities of experience—sensoriums through which the world is made and known
>
> —Matthew Fuller

Once software has been written and compiled, it is made to do work in the wider world; it not only represents the world but participates in it (Dourish 2001). Programs are run in order to read capta, process commands, and execute routines—to compile, synthesize, analyze, and execute. In common with earlier technological enhancement like mechanical tools or electrically-powered motors, software enjoys all the usual machine-over-man advantages in terms of speed of operation, repeatability, and accuracy of operation over extended durations, cost efficiencies, and ability to be replicated exactly in multiple places. Software thus quantitatively extends the processing capacity of electromechanical technologies, but importantly it also qualitatively differs in its capacity to handle complex scenarios (evaluating capta, judging options), taking variable actions, and having a degree of adaptability. Adaptability can be in terms of flexibility (making internal choices) but also degrees of plasticity (responding to external change). Software can also deal with feedback, or being able to adjust future conduct on the basis of past performance. In terms of adaptability, resistance to entropy, and response to feedback, it is clear that software truly makes the “universal machine,” in Turing’s famous phrase.

As Mackenzie (2006) notes, software is often regarded as possessing secondary agency—that is, its supports or extends the agency of others such as programmers, individual users, corporations, and governments; it enables the desires and designs of absent actors for the benefit of other parties. It does more than that though, in that software, like many other technologies, engenders direct effects in the world in ways never envisaged or expected by their creators and in ways beyond their control or intervention. Code also extends the agency of other machines, technical systems, and infrastructures. This is the case even if these effects are largely invisible from those affected, or where an effect is clear but not the executive role of software behind it. For example, code makes a difference to water supply infrastructure, being used to monitor demand quality and actively regulate flow rates to ensure acceptable pressure in the pipes. Turning the tap therefore indirectly but ineluctably engages with software, though the infrastructure appears dumb to the consumer who simply sees flowing water. In other cases, the elevator arrives, the car drives, the mail is delivered, the plane lands, the supermarket

shelves are replenished, and so on. Software divulges and affords agency, opens up domains to new possibilities and determinations. In other words, in Latour's (1993) terms, *software is an actant in the world* ; it possesses agency, explicitly shaping to varying degrees how people live their lives. **(p.40)**

Drawing on Ullman's ethnographic work, Fuller (2003, 31) makes the case that code enacts its agency through the production of events—blips—some outcome or action in an assemblage that the software contributes to, "the interpretative and reductive operations carried out on lived processes." To use Ullman's (1997) example further, in the context of being paid, the transfer of money from employer to employee is a blip in the relation between a company's payroll application and a bank's current accounts' software system. Here, a blip is not a signifier of an event, it is part of the event; the electronic transfer of funds that is then concretely expressed on the employee's bank balance. Fuller (2003, 32) argues that "these blips, these events in software, these processes and regimes that data are subject to and manufactured by, provide flashpoints at which these interrelations, collaborations and conflicts [between various agencies and actors such as employer, bank, tax agency, employee, etc.] can be picked out and analyzed for their valences of power, for their manifold capacities of control and production, disturbance and invention." In other words, for Fuller, we can start to chart the various ways in which software makes a difference to everyday life by studying where these blips occur—how capta is worked upon and made to do work in the world in various ways; to make visible the "dynamics, structures, regimes, and drives of each little event … to look behind the blip" (2003, 32). Moreover, these blips are contextual and signifiers of other relations—for example, a person's bank balance is an instantiation of class relations—and they themselves can be worked upon and reinterpreted by code to invent a sequence of new blips (Fuller 2003). Of course, this is not always easy, especially as software is designed to run silently in the background; to be inscrutable; and its use and conventions to appear rational, logical, and commonsensical. It seeks to slide beneath the surface because designers aim for people to be interfacing with a task, not with a computer (see figure 2.3).

As Fuller notes, implicit in the notion of software as actant is power—the power to shape the world in some way. For us, this power needs to be understood as relational. Power is not held and wielded by software; rather, power arises out of interrelationships and interactions between code and the world. From this perspective, code is afforded power by a network of contingencies, but in and of itself does not possess power (Allen 2004). In this sense, Lessig's (1999) assertion that "code is law" is only partially correct. Code might well follow certain definable architectures, defaults, and parameters within the orbit of the code itself—how the 1s and 0s are parsed and processed—but their work in the world is not deterministic as we discuss at length in chapter 5. In other words, code is permitted to express certain forms of power (to dictate certain outcomes) through the channels, structures, networks, and institutions of societies and permissiveness of those on whom it seeks to work, in the same way that an individual does not hold and wield power but is afforded it by other people, communal norms, and social structures. Of course, it should be acknowledged that people are worked upon by expressions of power not of their **(p.41)**

choosing or consent, but such power is always relational in nature. What that means for software is, as Mackenzie (2006, 10) notes, “agency distributes itself … in kaleidoscopic permutations.”

One of the effects of abstracting the world into software algorithms and data models, and rendering aspects of the world as capta, which are then used as the basis for software to do work in the world, is that the world starts to structure itself in the image of the capta and code—a self-fulfilling, recursive relationship develops. As Ullman (1997, 90) notes, “finally we arrive at a tautology: the [cap]ta prove the need for more [cap]ta! We think we are creating the system, but the system is also creating us. We build the system, we live in its midst, and we are changed.” For example, because software can undertake billions of calculations in a very short space of time, it can undertake analytical tasks that are too computationally demanding for people **(p. 42)** to perform manually. In so doing, it can reveal relationships about the world that would have otherwise remained out of view. An apposite case of this is in academia, across both the hard sciences and in the humanities, where computers are enabling innovative forms of analysis, new theories, and new inventions. These in turn discursively and materially reshape our world, as the example of climate modeling earlier demonstrated. Further, as we detail in the following chapters, the way we work, consume, travel, and relax have been reconfigured with respect to the possibilities that software has offered.

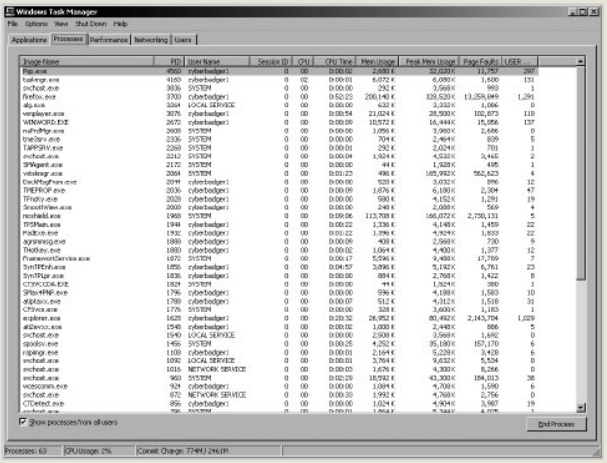


Figure 2.3 “You think you know your computer, but really all you know is a surface on your screen” (Annette Schindler quoted in Mirapaul 2003). The numerous software processes running on one of the authors’ laptops as he edits this chapter. Most are performing work of unknown importance to the immediate tasks at hand.

For Mackenzie (2002) the reason why software can do work in the world is because it possesses *technicity* . Technicity refers to the extent to which technologies mediate, supplement, and augment collective life; the unfolding or evolutive power of technologies to make things happen in conjunction with people. For an individual technical element such as a tool like a carpenter’s saw, its technicity might be its hardness and flexibility (a product of human knowledge and production skills) that enables it in conjunction with human mediation to cut well (note that the constitution and use of the saw is dependent on both human and technology; they are inseparable). As Star and Ruhleder (1996, 112, our emphasis) note, a “tool is not just a thing with pre-given attributes frozen in time—but a thing becomes a tool *in practice* , for someone, when connected to some particular activity.… The tool emerges in situ.” In other words, a saw is not simply given as a saw, rather it becomes a saw through its use in cutting wood. Similarly, a shop emerges as a shop by selling goods to customers. In large scale ensembles such as a car engine, consisting of many components, technicity is complex and cannot be isolated from the sum of individual components (and their design, manufacture, and assembly), its “associated

milieu" (for example, the flow of air, the lubricants, and fuel), and its human operator, "that condition and is conditioned by the working of the engine" (Mackenzie 2002, 12).

Software possesses high technicity; it is an actant able to do work in the world, enabling everyday acts to occur such as watching television, using the Internet, traveling across a city, buying goods, making phone calls, and withdrawing money from an ATM. In these cases, and myriad other everyday tasks, the software acts autonomously, and at various points in the process is able to automatically process inputs and to react accordingly to solve a problem. While some of these practices were possible before the deployment of software, it is now vital to their operation. The technicity of code in such cases is high as they are dependent on software to function with no manual alternatives. As already noted, the technicity of code is not deterministic (for example, code turns everyday practices into absolute, non-negotiable forms) or universal (which is to say, such determinations occur in all places and at all times in a simple cause-and-effect manner). Rather, technicity is contingent, negotiated, and nuanced; realized through its practice by people in relation to historical and geographical context. As such, there is no neat marriage between coded objects, infrastructures, **(p.43)** processes, and assemblages and particular effects of code. Instead, technicity varies as a function of the nature of code, people, and context.

For example, technicity varies depending on the autonomy and consequences of software. Autonomy relates to the extent to which code can do its work without direct human oversight or authorization. The degree of autonomy is a function of the amount of input (the system's knowledge of its environment and memory of past events), sophistication of processing, and the range of outputs that code can produce. If code crashes, then the consequences of its failure can range from mild inconvenience (such as travel delays) to serious economic and political impacts (such as the failure of the power grid), to life-threatening situations (when vital medical equipment is unable to function or air traffic control towers are unable to direct planes). All types of software do not have the same social significance. For example, the technicity of the host of codes found in a typical home is radically different from that employed in a hospital intensive care unit.

Further, the technicity of code varies according to the people who are entangled with it. Not all people experience or interact with the same code in the same way. Many factors influence a person's interaction with code, from personality and social characteristics (such as age, gender, class, and ethnicity), to economic or educational status, personal histories and experiences, their intentions, their technical competencies, and even whether they are on their own or in a group. Software and its effects vary across individuals. For example, someone with daily use of a software system may experience it in a more banal and ambivalent way than a person encountering it for the first time. As noted above, the relationship between code and people also varies as a function of wider context, including crucially the places where it is occurring. Interactions with code are historically, geographically, and institutionally embedded, and do not arise out of nowhere. Rather, code works within conventions, standards, representations, habits, routines, practices, economic situations, and discursive formations that position and place how code engages and how code is engaged. The use of code is then always prefaced by, and contingent upon, this wider context.

Conclusion

In this chapter we have detailed how we understand and conceptualize the nature of software. Code is an expression of how computation both capture the world within a system of thought (as algorithms and structures of capta) and a set of instructions that tell digital hardware and communication networks how to act in the world. For us, software needs to be theorized as both a contingent product of the world and a relational producer of the world. Software is written by programmers, individually and in teams, within diverse social, political, and economic contexts. The production of **(p.44)** software unfolds—programming is performative and negotiated and code is mutable. Software possesses secondary agency that engenders it with high technicity. As such, software needs to be understood as an actant in the world—it augments, supplements, mediates, and regulates our lives and opens up new possibilities—but not in a deterministic way. Rather, software is afforded power by a network of contingencies that allows it do work in the world. Software transforms and reconfigures the world in relation to its own systems of thought. Thinking about software in these terms allows us to start to critically think through its nature and to consider where and how it works. In part II of the book we expand this analysis to think through the difference that software makes, examining how code alters the nature of objects, affects how space is transduced, changes how societies are governed, and even how software is affording new kinds of creativity and empowerment. What our analysis makes clear is that software as an actant is radically transforming our world, and that software itself, and not simply the technologies it enables, merits significant intellectual attention.