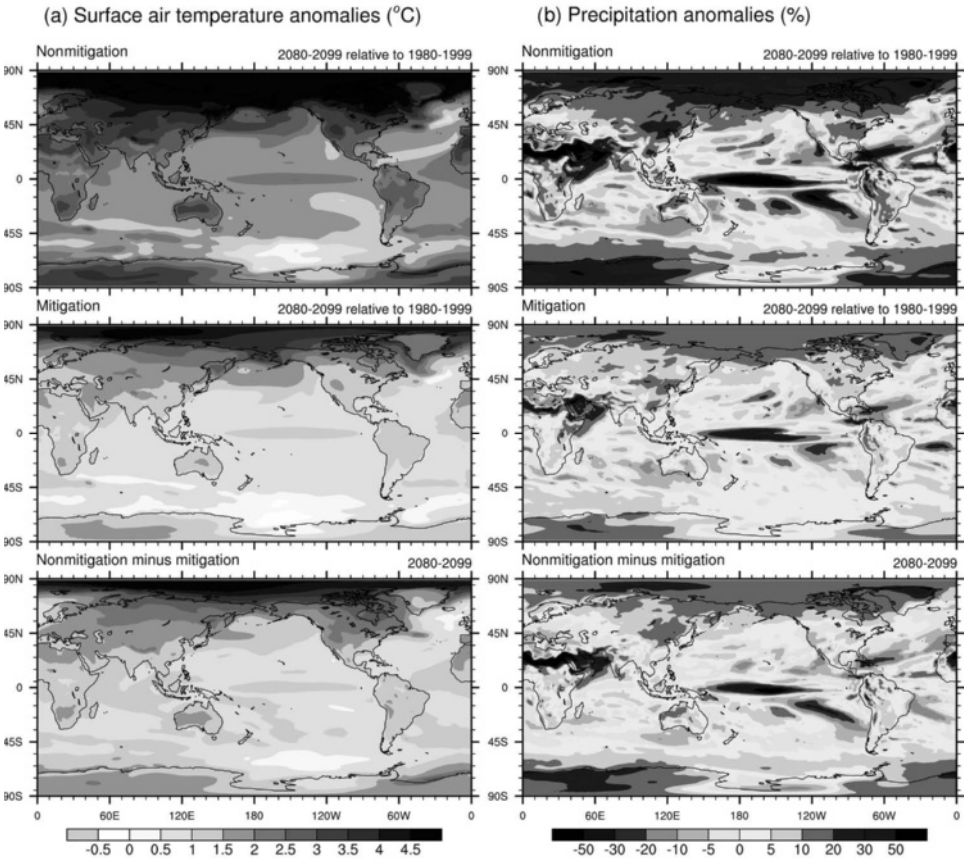


Figure 2.2

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.)

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.

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

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

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 com-

ponent 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, Sourceforge 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

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, contextualization 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

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

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.

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.

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

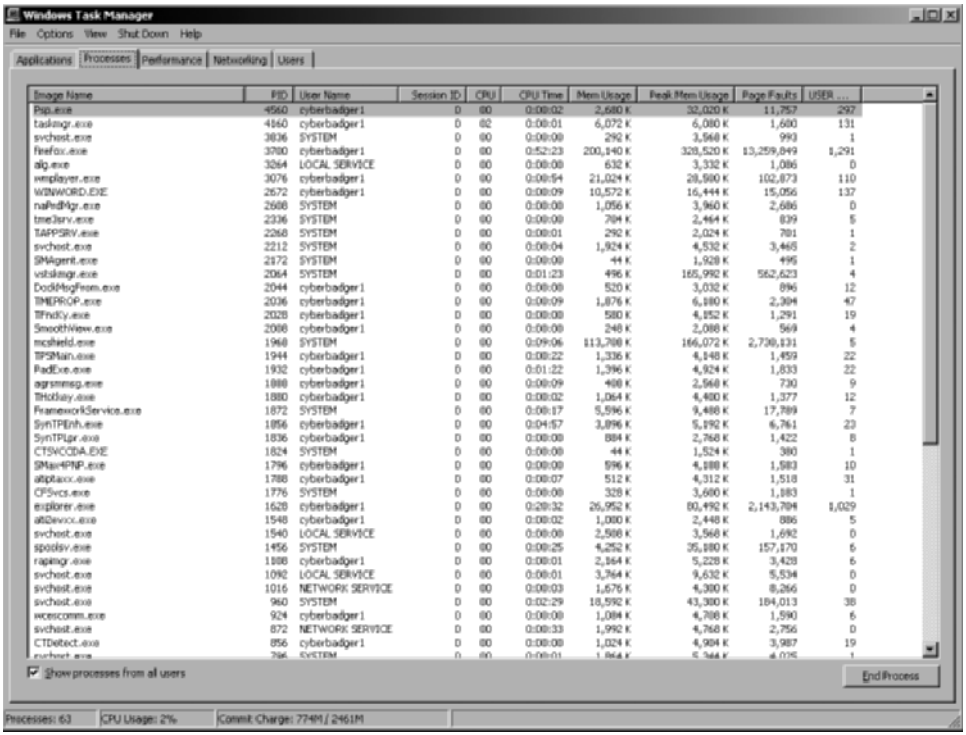


Figure 2.3
"You think you know your computer, but really all you know is a surface on your screen" (Annette Schindler quoted in Mirapaul 2003). The numerous software processes running on one of the authors' laptops as he edits this chapter. Most are performing work of unknown importance to the immediate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

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

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.

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, infra-

structures, 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

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.

II The Difference Software Makes

3 Remaking Everyday Objects

When it is not only "us" but also our "things" that can upload, download, disseminate and stream meaningful-making stuff, how does the way in which we occupy the physical world become different?

—Julian Bleecker

When each object has a unique identity, objects begin to seem more like individuals, and individual people become susceptible to being constituted as objects.

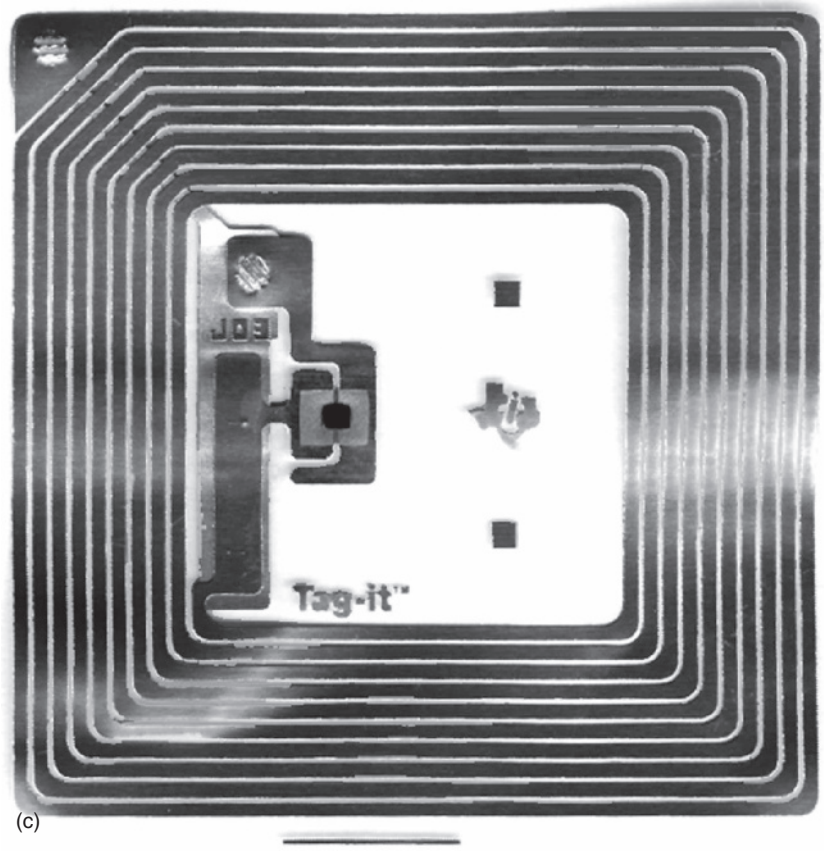
—N. Katharine Hayles

So far we have contended, in broad terms, that software makes a difference to everyday life because it possesses technicity. In this chapter, we look at how the nature of material objects and the work they do has been transformed by code. Software, as we detail below, is imbuing everyday objects, such as domestic appliances, handheld tools, sporting equipment, medical devices, recreational gadgets, and children's toys, with capacities that allow them to do additional and new types of work. On the one hand, objects are remade and recast through interconnecting circuits of software that makes them uniquely addressable and consistently machine-readable, and thus exposed to external processes of identification and linkage that embeds them in the emerging "Internet of things" (in much the same way that the location of a web site can be looked up through its unique domain name from anywhere on the Internet, it is envisaged that this infrastructure will facilitate the same for any uniquely tagged object; Schoenberger 2002). On the other hand, software is being embedded in material objects, imbuing them with an awareness of their environment, and the calculative capacities to conduct their own work in the world with only intermittent human oversight, to record their own use, and to take over aspects of decision making from people. In so doing, our approach is one of building a taxonomy that classifies new types of coded objects as a way to start to make sense of how objects are becoming addressable, aware, and active.

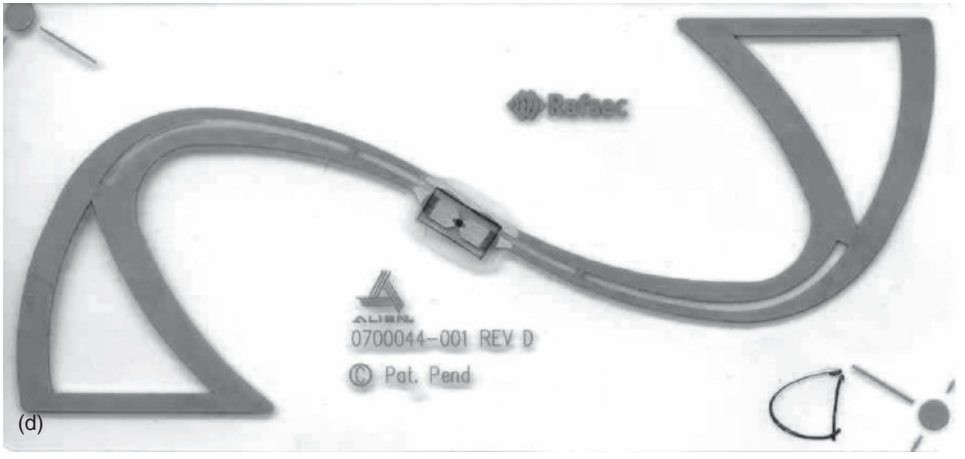
Making Objects Addressable

Since the late 1970s and the widespread application of bar codes to mass-produced consumer goods, objects have increasingly become machine-readable through rapid and reliable reading of identification numbers placed on them (see figure 3.1). Such identification technologies include a range of different printed bar codes, the growing use of radio frequency ID (RFID) tags, magnetic strips, embedded chips, and transponder units which, when read and combined with appropriate information infrastructure (for identification number allocation and specifying product classification formats), can be consistently matched to information held in an organization's captabase to reveal the identity of the object and other associated properties (such as batch number, date of manufacture, and shipping history). As a result, while different instances of the same class of product would have previously shared the same bar code identifier (say bottles of whiskey), now each and every instance (bottle) is uniquely indexable and can be tracked through time and space in ways that were previously impossible. What this increased granularity of addressing means is a twofold change in the status of objects. First, the ontological status of each object is uniquely indexed. This information is knowable in new ways in terms of what information is attributable to the object, and can be generated with respect to it—ranging from purchasing information through to a detailed usage trail and eventual disposal (so it is not just *a* bottle of whiskey in a household's trash can but *the* bottle of whiskey purchased for $19.99 on 12/10/2009, at 19:54 p.m., in the Whistlestop convenience store, Downtown Crossing, by A. N. Other). Second, individuated identification transforms the epistemological

Figure 3.1
The physical manifestation of identification systems. (*a, b*) Bar codes that can be read visually. (*c, d*) RFID chips that can be queried remotely by radio signals.



(c)



(d)

Figure 3.1
(continued)

status of each object, with it being useable in new ways and able to do additional work in the world and to be worked upon by other entities such as information systems.

There are presently two classes of machine-readable objects: passive and active (see figure 3.2), based on the technicity of the tagging technology that holds the unique identification number. Passive machine-readable objects have to be directly queried (for example, by laser scanner) to capture their identification details. In contrast, active ID tags continually broadcast their details. For example, an active RFID tag consists of a simple digital circuit into which data are embedded, an antenna which broadcasts the information, and a battery to power the unit. These tags can be read at a distance by a radio transponder, rather than having to be passed in line of sight to a laser scanner. In both cases, passive and active, the data read or transmitted is one-dimensional and invariant (typically only a unique identification code number). The object itself does not generate and communicate new capta about its status. However, because the captured information is usually transmitted through and queried across distributed networks, machine-readable objects become active constituent parts of circuits of interchange between objects, sensors, captabases, software algorithms, and work.

Let us consider RFID tags in more detail. The industry aim of RFID tags is to provide a means to automatically identify any object, anywhere. They are presently most widely used in vehicle dashboard tags for automatic toll payment (a widespread system in the United States being E-ZPass) and in livestock to facilitate "farm-to-fork" traceability given growing concerns over food safety (Popper 2007). The main application of RFID tags is likely to be in retail, where they are seen as a major advance in inventory management, the fight against shoplifting, staff pilfering, and in enhancing

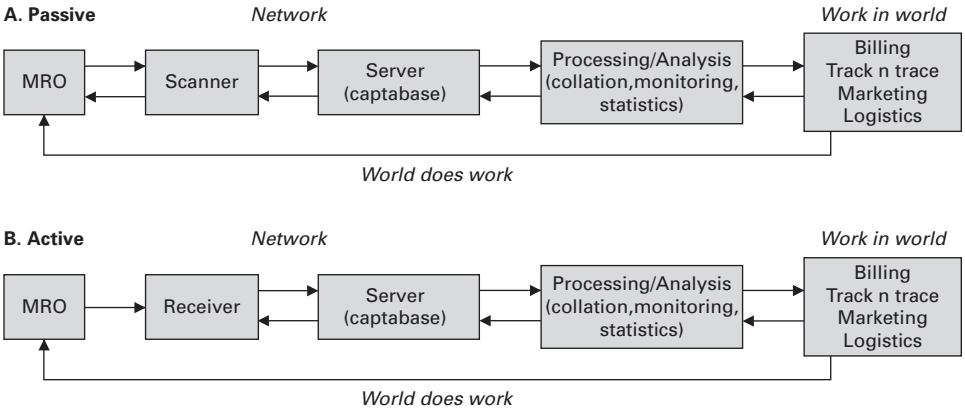


Figure 3.2
Classes of machine-readable objects.

customer profiling (Ferguson 2002) and in surveillance and security (for example, RFID tags are embedded into new passports; see chapter 5).

The leading standard with regard to uniquely coding objects with RFID tags is known as the Electronic Product Code (EPC), developed by the Auto-ID Center, an industry-sponsored R&D lab at MIT, and commercially implemented by EPCglobal Inc. (a joint venture of the Uniform Code Council and EAN International, the main players in UPC bar code management). Through EPCs, RFID tags can be linked together into a global information network—an Internet of things—which provides the means to automatically look up details on any tagged object from any location across distributed networks. Borrowing the domain name schema used on the Internet, the EPC network uses a distributed Object Naming Service (ONS) to link each EPC number to an appropriate naming authority database that provides detailed information. Importantly, the querying of the ONS as RFID tagged products move through supply chains will automatically create richly-detailed audit trails of capta. The result will be a much greater degree of routine machine-to-machine generated knowledge on the status and positioning of many millions of physical objects in time and space.

Evangelists for RFID tagging and the EPC network envision a wide range of innovations in the handling of physical objects arising from these networked capta trails (see Ferguson 2002, for typical speculation). Inside retail stores, a key aim is so-called smart shelving (units that are aware of their own stock levels and the activities of individual customers). In the home, pundits are predicting microwave ovens that check and set the best cooking settings for frozen dinners, washing machines that read the tags of clothing to automatically select the most appropriate wash cycle, and medicine cabinets able to spot out-of-date or recalled pharmaceuticals. There could also be potential for tracking goods at the end of the life cycle, alerting waste disposal collectors to items containing toxic substances, for example. Yet bound up with the promises of greater convenience and more orderly domestic routines, is the capacity to make formally hidden and unrecorded actions newly visible to external organizations and to eliminate anonymity from mass consumption because every time an RFID tag is queried it leaves behind a log. As such, RFID smart tagging raises the specter of a new frontier of potentially invasive surveillance (Albrecht and McIntyre 2005; van Kranenburg 2008; see chapter 5). It also opens up a significant debate about property rights (Hayles 2009) in that a person may legally own a material object but not possess the capta that is generated with respect to it. And such "virtualized [cap]ta about the object has market values that amount to considerable percentages of the value of the material commodities to which the [cap]ta correspond" (Hayles 2009, 54).

Regardless of their wider uses, it is likely that over the next few years RFIDs will replace bar codes on retail packaging and be embedded in all manner of manufactured goods to facilitate asset management and logistics. It is already fair to say, as we explore in chapter 9, that as part of larger infrastructures of identification and addressing, RFID


Vaillant
Th. 30.08.07
10:15
15.0 °C Outside
Auto
21.0 °C
VRC 430
+
−

(a)


37.0 °C
PHILIPS
AVENT

(b)

Figure 3.3

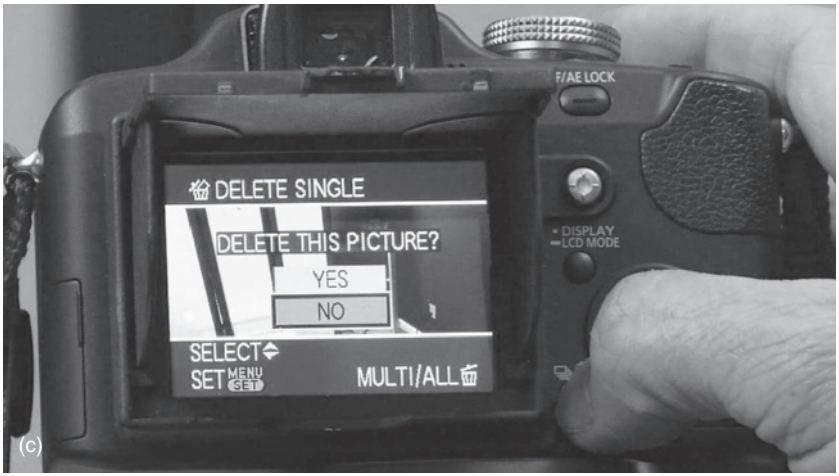


Figure 3.3
A range of coded objects used to solve domestic tasks of heating, health and well-being, capturing photograph memory card data, and cleaning of clothes. The presence of digital display screens is an indicator of the executive presence of software. (*a*) Thermostat (*Source*: www.vaillant.co.uk). (*b*) Pacifier with a built-in thermometer (*Source*: www.p4c.philips.com/files/s/sch540_00/sch540_00_pss_eng.pdf). (*c*) Digital camera interface. (*d*) Washing machine interface.

tagging systems have already started to reshape modes of production and the processes of capital accumulation on a variety of scales. They also pose a significant intellectual and political challenge (see chapter 5).

Coded Objects

In contrast to machine-readable goods and products which simply participate externally in the Internet of things, some objects have code physically embedded into their material form, altering endogenously their ongoing relations with the world. In such objects, software is used on the one hand to enhance the functional capacity of what were previously dumb objects, enabling them to sense something of their environments and to perform different tasks, or the same tasks more efficiently (figure 3.3), or to be plugged into new distributed networks that afford some value-added dimension such as data exchange on how they are used. On the other hand, code is used to underpin the design and deployment of new classes of objects, particularly mobile devices (such as PDAs, MP3 players, and GPS), that in some cases replace analog equivalents (diaries and date books, portable cassette tape and CD players, paper maps and guide books) or undertake entirely new tasks. In either case, the embedding of software significantly increases an object's technicity.

In thinking through the relationship between code and its embedding into objects, we have used the decision-making process detailed in figure 3.4 to subdivide coded objects into two general types based on the level of significance of software to an object's primary function(s). *Peripherally coded objects* are objects in which software has been embedded, but such code is not essential to their use (that is, if the software fails, they still work as intended, but not as efficiently, cost-effectively, or productively). *Codejects* on the other hand are dependent upon code to function—the object and its code are thoroughly interdependent and inseparable (hence our conjoining of the terms code and object to denote this mutual interdependence). Codejects can be further subdivided into three main classes on the basis of the following characteristics: their programmability, interactivity, capacity for remembering, their ability for anticipatory action in the future based on previous use, and relational capacities. In summary, these classes are hard codejects, unitary codejects, and logjects.

Hard codejects rely on code to function but are not programmable and therefore have low levels of interactivity.

Unitary codejects are programmable, exhibit some level of interactivity (although this is typically limited and highly scripted), and do not record their work in the world. They can be subdivided into two groups: (a) *closed codejects* and (b) *sensory objects* depending on whether they sense and react to the world around them.

Logjects are objects that have an awareness of themselves and their relations with the world and which, by default, automatically record aspects of those relations in logs

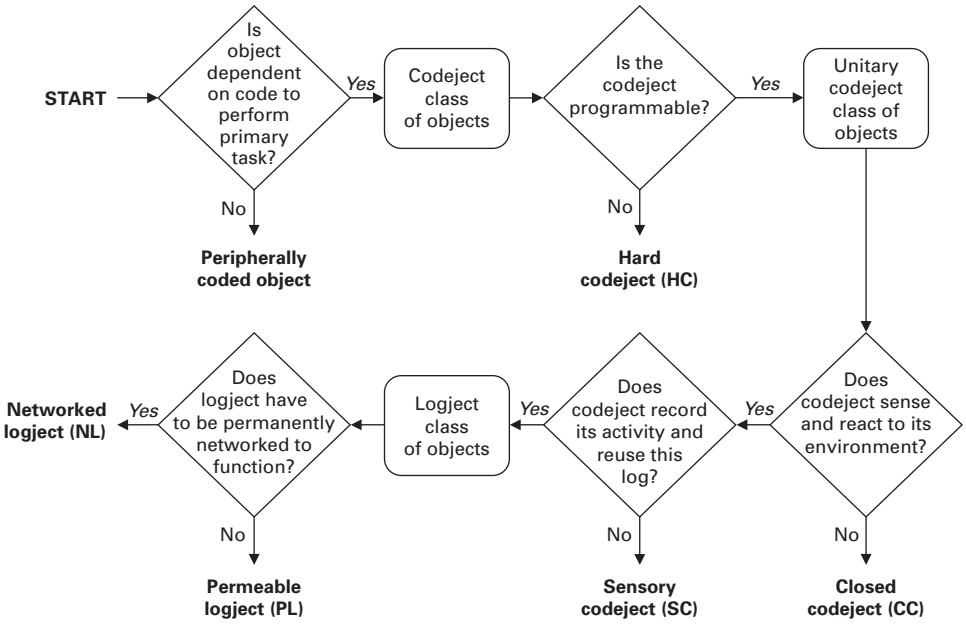


Figure 3.4
A decision tree of the characteristics of different classes of coded objects.

that are stored and reused in the future. Logjects often have high levels of interactivity and multifunctionality. Logjects can also be subdivided into two groups based on their capacity to work independently of wider networks: (a) *permeable logjects* and (b) *networked logjects*.

Peripherally Coded Objects

Peripherally coded objects are objects in which software has been embedded, but where the software is incidental to the primary function of the object. There are relatively few such objects and, in most cases, the code merely augments use, but is by no means essential to its functioning. Often, the presence of code is merely an adornment that serves the purpose of product marketing, to differentiate it from predecessors or acts as a token of added value. For example, an oven might have a digital timer embedded in it, but if this ceases to function, then the appliance will continue to cook food. Similarly, an exercise bike might have a device that digitally displays the speed at which the cyclist is pedaling, but if this ceases to work, the equipment still enables exercise to take place. In both cases, the code does little more than augment the object's use by enabling the chef to know how long a dish has been cooking and the cyclist to know the traveling speed. Both are simply digital replacements for analog technology.

Hard Codejects

Hard codejects have a special kind of software, known as firmware, embedded into them that is essential for their functioning. Firmware consists of a defined set of routines being stored permanently in read-only memory on a device, rather than being enacted through an executable program that can be accessed and interfaced with. Examples include a USB memory stick and basic SCADA (supervisory control and data acquisition) systems wherein code monitors and controls an element of infrastructure, such as a valve. Both rely on code, but its functionality is predetermined and fixed.

Unitary Codejects

Unlike hard codejects, unitary codejects are programmable to some degree and therefore exhibit some degree of interactivity; users are able to control some aspects of the object's functionality, instructing it as and when required. They, along with logjects, exhibit *liveness*—a feeling that there are infinite possibilities to explore; *plasticity*—the person interacting with the codeject feels that they can push its limits without breaking the system; *accretion*—the computation improves and evolves with use; and *interruption*—computation is open to unpredictable input and can react to it without breakdown (Murtaugh 2008).

We term them *unitary* because they are self-contained, having everything they need to function within their material form. In broad terms, unitary codejects can be divided into those that function independent of their surroundings (closed codejects), and those that are equipped with some sensors that enable the object to react meaningfully to particular variables in their immediate environment (sensory codejects).

Closed codejects can include digital clocks, and some audiovisual equipment such as radios, CD players, and DVD players. Code is vital to the functioning and performance of each of these items, but the object executes its task independent of the world around it. Each is programmable to some degree—the time can be adjusted, stopwatch operated, alarm and record times set, the order of tracks selected—but generally they have circumscribed functions and limited latitude to operate automatically.

Sensory codejects have some awareness of their environment and react automatically to defined external stimulus: common domestic examples include a heating/air conditioner control unit; a washing machine that is monitored and controlled by software; and a digital camera and storage card. The heating/air conditioning unit is equipped with a digital thermostat and timer that is aware of the time/date and senses the surrounding temperature. Simple software algorithms react accordingly to temperature measurements in relation to preset requirements. Similarly, software embedded in the washing machine will monitor multiple contextual parameters such as door lock, load weight, and water temperature, necessary for safe and effective operation without human oversight (see table 3.1). The digital camera captures an image of the

Table 3.1
Hotpoint washer-dryer error codes that are displayed by software to the user

Fault codes for LCD EVO1 Washing Machines and Washer Dryers
* F01—Short circuit motor triac—Book a washing machine repair.
* F02—Motor jammed tacho detached—Book a washing machine repair.
* F03—Wash thermistor open/short circuit—Book a washing machine repair.
* F04—Pressure switch jammed on empty—Book a washing machine repair.
* F05—Pressure switch jammed on full—Book a washing machine repair.
* F06—Program selector error—Book a washing machine repair.
* F07—Heater relay stuck—Book a washing machine repair.
* F08—Heater relay cannot be activated—Book a washing machine repair.
* F09—Incompatible eeprom—Book a washing machine repair.
* F10—Pressure switch not sensing correctly—Book a washing machine repair.
* F11—Pump cannot be activated—Book a washing machine repair.
* F12—Communication error—Book a washing machine repair.
* F13—Dryer fan or dryer thermistor faulty—Book a washing machine repair.
* F14—Dryer element faulty—Book a washing machine repair.
* F15—Dryer element relay faulty—Book a washing machine repair.
* H20—Not fillings. Check tap, hose and inlet valves.
* LOCKED—Check interlock—Book a washing machine repair.

Note: These codes give a partial indicator of the range of conditions that the appliance's software monitors.
Source: Hotpoint Service web site "Error Messages and Error Codes," www.hotpointservice.co.uk/hs/pages/content.do?keys=FAQ:ERROR_CODES

world using a CCD sensor and measures light levels to adjust the aperture setting and the lens movement for auto-focusing, as well as monitoring the remaining battery life and available storage space.

Logjects

Logjects differ from unitary objects in that they also record their status and usage, and, importantly, can retain these logs even when deactivated and utilize them when reactivated. In key ways, these logs can have a bearing on the ongoing operation of the object and its relations with people or wider processes. Furthermore, part of their functionality is externalized, lying beyond the immediate material form of the object.

We derive the term *logjects* from Bleecker's (2006) notion of a blogject (where for us a blogject is one type of logject). Bleecker defines a *blogject* as an emerging class of software-enabled objects that generates a kind of blog of its own use and has the capability to automatically initiate exchanges of socially meaningful information—"it is an artifact that can disseminate a record of experience to the web" (Nova and

Bleecker 2006, no pagination). Bleecker (2006, 6, original emphasis) characterizes blogjects as objects that: (1) can "track and trace where they are and where they've been," (2) "have self-contained (embedded) histories of their encounters and experiences" (rather than indexed histories), and (3) "have some form of agency—they can foment action and participate; they have an assertive voice within *the social web*." Blogjects are things that can do meaningful social acts where their actions shape how people think about and act in the world; they "participat[e] within the Internet of social networks" (Bleecker 2006, 2). Here, Bleecker is very much interested in only a certain kind of software-enabled object, those that produce streams of information very much like a blog written by a person, thus contributing to the "ecology of networked publics—streams, feeds, trackbacks, permalinks, wiki inscriptions and blog posts" (Bleecker 2006, 9). He is very careful to delineate blogjects as political actants that contribute to debates by providing socially meaningful information, rather than being coded objects that log their use and communicate and/or analyze that data across distributed networks.

While Bleecker's notion of a blogject has conceptual utility, for us, it is one form of a logging object in a much larger sociotechnical ecology of logjects. We broadly define a logject as an object with embedded software able to monitor and record, in some fashion, its own operation. More specifically, and expanding on Bleecker, a logject has the following qualities:

- It is uniquely indexable.
- It has awareness of its environment and is able to respond meaningfully to changes in that environment within its functional context.
- It traces and tracks its own usage in time and/or space records that history, and can communicate that history across a network for analysis and use by other agents (objects and people).
- The logject can use the capta it produces to undertake what we have previously termed automated management (Dodge and Kitchin 2007a)—automated, automatic, and autonomous decisions and actions in the world without human oversight and to effect change through the "consequences of their assertions" (Bleecker 2006, 9).
- A logject is programmable and thus mutable to some degree (that is, it is possible to adjust settings, update parameters, and download new firmware).

Logjects then enable the kinds of unobtrusive machine-to-machine, machine-to-person, and person-to-machine exchanges that are a fundamental trait of pervasive computing and are diverse in nature.

Permeable Logjects Permeable logjects consist of relatively self-contained units such as an MP3 player, a PDA, and a GPS, all of which have the potential to be connected to wider networks. Such devices trace and track their usage by default, recording this capta as an embedded history; are programmable in terms of configurable settings and

creating lists (for example, they play lists of songs, keep calendar entries, and provide route itineraries); perform operations in automated, automatic and autonomous ways; and engender socially meaningful acts such as entertaining, remembering an important meeting, and helping an individual to travel between locations. These devices work to relieve the cognitive burden of routine tasks on people who use them, and help to reduce the risks and consequence of unexpected events. Unlike a networked logject, all essential capacities are held locally, and primary functionality does not require a network connection to operate. That said, appropriate capta (such as music, calendar entries, or map files) and software must be downloaded onto the machine at some point; hence they are permeable. These devices can be connected to wider networks in order for information to be uploaded and exchanged with other devices (via Bluetooth wireless transmission, for example) and updates in firmware can be downloaded, though typically this is not automatic and sometimes requires considerable human intervention (what might be classed as digital housework, for example, syncing a PDA or MP3 player). The uploaded information can be processed and analyzed in relation to other usage, thus providing added value. The aggregate social significance of such objects is impossible to estimate, but they are used to solve all manner of domestic problems billions of times a day, often without the active awareness or involvement of people.

Networked Logjects Networked logjects do not function without continuous access to other technologies and networks. In particular, because they need constant two-way data exchanges, they are reliant on access to a distributed communication network to perform their primary function. Such logjects track, trace, and record their usage locally, but because of memory issues, the necessity of service monitoring/billing, and in some cases a user's ability to erase or reprogram such objects, their full histories are also recorded externally to an immediate material form. Some networked logjects are relatively fixed in the environment (satellite/cable television control boxes, home security monitoring systems) and others are inherently mobile (cell phones, vehicles with remote monitoring) that use a range of communication technologies such as GSM, Wi-Fi, and Bluetooth to maintain a network connection. Mobile networked logjects continuously search for connectivity and can respond automatically and autonomously to the network conditions. For example, a cell phone reacts automatically to incoming calls by sounding the ringtone, switches to the answer service if the call is unanswered, and alerts the owner that a call was missed and/or a message is waiting.

Objects Become and Do Other Things

Reflecting on this taxonomy, and the ways in which objects are becoming either externally machine-readable or endogenously coded, it seems to us that the nature of

many objects and the material processes that constitute everyday life are being remade in quite radical ways—objects are being alternatively reconfigured and defined, they are gaining additional capacities to do additional work in the world, and the world can do more work on and through them. Individual objects are now knowable in new ways—they are uniquely identifiable and their use and movement is trackable across space and time. They are becoming part of the emerging Internet of things. Objects thus gain capta shadows that can be analyzed for emergent properties, with new knowledge of the life of an object used to refine the system through which it is made, distributed, sold, and potentially used. And importantly, such capta can be processed to anticipate potential future activities.

Let's explore the example of the archetypal machine-readable object: a credit card. A credit card has a machine-readable unique identifier embedded in its chip and/or magnetic strip. Functionally, it is a conveniently-sized material token used to authenticate access to transaction records in financial captabases. The card can be enrolled by software into a secure communications process with the financial captabase through an intranet, which also tracks spending across time and space and is increasingly subjected to real-time analysis for fraud detection. The capta gathered from the card can also be aggregated to provide an individualized spatial history of consumption. In turn, a plethora of other material objects and sociotechnical infrastructures have been built around it (from the design of ATMs and card readers, online baskets, and SSL encryption, through to the mundane shape of wallets). Through its membership of the Internet of things, the credit card does work in the world—enabling the purchasing of goods and services more securely and efficiently than in previous incarnations, and it is also worked upon, with the information concerning usage employed to evaluate credit levels and anticipate future risk; to monitor purchases in real time for potential fraud and regulate the amount of spending for accurate billing; as well as feeding some of the capta into wider marketing profiles and geodemographics models (Burrows and Gane 2006; Leyshon and Thrift 1999; see chapter 9). Importantly, it can legally, and through social convention, now hold a measure of trust that allow actions at a distance that are replacing embodied transactions (Lyon 2002).

Now let us examine the example of a codeject: a cell phone. A cell phone is wholly dependent on software to function; without the ongoing updates of code it cannot work; it is an inert piece of plastic and other materials. Software enables the phone to act as a traditional phone, but to also undertake a range of additional functions such as the sending and receiving of texts, music, and other information; connecting to the Internet; to take and send photos; act as an alarm clock or calendar or a radio or a recorder or a calculator; to play games; to store phone numbers, images, and other files; and so on. It enables the phone to be customizable, selecting ring tones, wallpaper, profiles, and connectivity type. In other words, a cell phone enables its user to do a range of different activities. And fundamentally it allows the user to do all of