I Introduction

1 Introducing Code/Space

Software is everything. In the history of human technology, nothing has become as essential as fast as software.
—Charles Fishman

Our civilization runs on software.
—Bjarne Stroustrup

Over the past thirty years, the practices of everyday life have become increasingly infused with and mediated by software. Such are the capacities and growing pervasiveness of software that it has become the lifeblood of today's emerging information society, just as steam was at the start of the industrial age. Software, like steam once did, is shaping our world—from the launch of billion-dollar spacecraft to more mundane work such as measuring and displaying time, controlling traffic lights, and monitoring the washing of clothes. Indeed, whatever the task—domestic chores, paid work, shopping, traveling, communicating, governing, playing—software increasingly makes a difference to how social and economic life takes place. In short, software matters, and this book documents how and why it does. We detail how software produces new ways of doing things, speeds up and automates existing practices, reshapes information exchange, transforms social and economic relations and formations, and creates new horizons for cultural activity. And we do so by explicitly making software the focus of critical attention rather than the technologies it enables.

Software

As we explore in more detail in chapter 2, software consists of lines of code—instructions and algorithms that, when combined and supplied with appropriate input, produce routines and programs capable of complex digital functions. Put simply, software instructs computer hardware—physical, digital circuitry—about what to do (which in turn can engender action in other machinery, such as switching on electri-

cal power, starting a motor, or closing a connection). Although code in general is hidden, invisible inside the machine, it produces visible and tangible effects in the world.

Software is diverse in nature, varying from abstract machine code and assembly language to more formal programming languages, applications, user-created macros, and scripts. One way to consider these forms is as a set of hierarchically organized entities of increasing complexity that parallel that of organic entities (figure 1.1). Software takes form in the world through multiple means, including hard-coded applications with no or limited programmability (embedded on chips), specialized applications (banking software, traffic management systems), generic user applications (word processors, Web browsers, video games), and operating systems (Windows, Mac

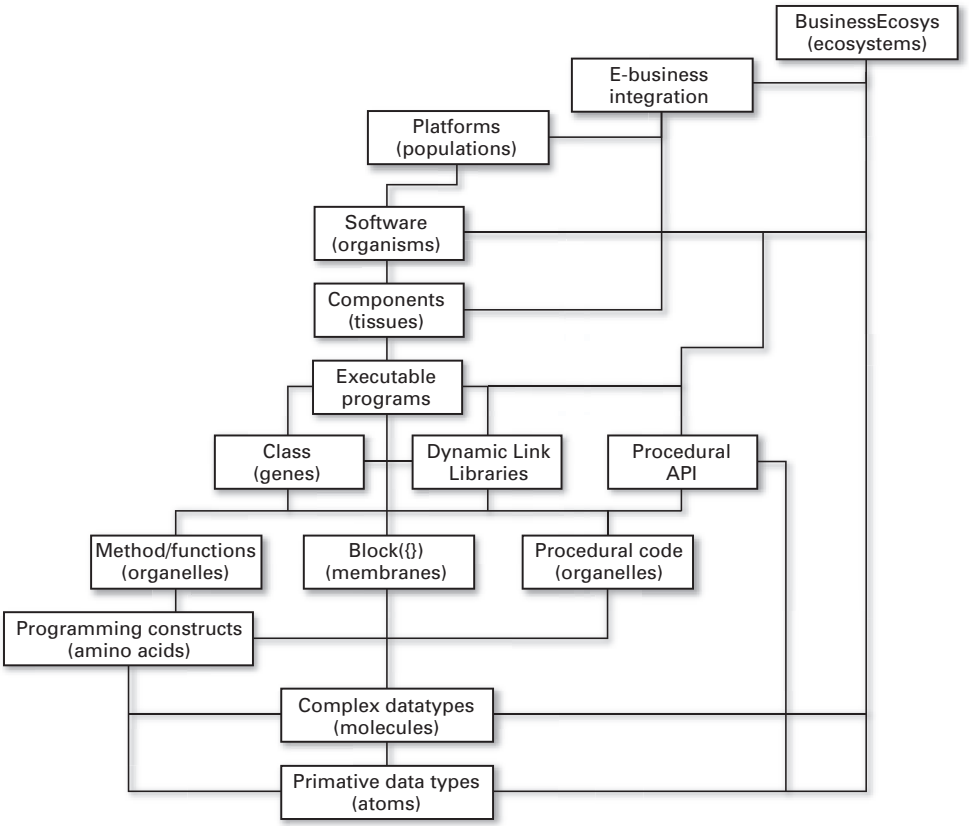


Figure 1.1
A hierarchical conceptualization of different scales of software. The approximate parallels with biological entities are indicated in parentheses. (Redrawn from Nguyen 2003, 10)

OS, Linux), that run on a variety of hardware platforms (embedded chips, dedicated units, PCs, workstations) and can generate, distribute, monitor, and process capta (capta are units that have been selected and harvested from the sum of all potential data, where data are the total sum of facts in relation to an entity; in other words, with respect to a person, data is everything that it is possible to know about that person, capta is what is selectively captured through measurement—see glossary) and information flows across a range of infrastructures (printed circuit boards, coaxial and fiber-optic cables, wireless networks, satellite relays) using a variety of forms (electrical, light, microwave, radio).

The phenomenal growth in software creation and use stems from its emergent and executable properties, that is, how it codifies the world into rules, routines, algorithms, and captabases (a collection of capta stored as fields, typically within a tabular form, that can easily be accessed, managed, updated, queried, and analyzed; traditionally named a database, it has been renamed to recognize that it actually holds capta not data) and then uses these to do work in the world. Although software is not sentient and conscious, it can exhibit some of the characteristics of being alive. Thrift and French (2002, 310) describe it as "somewhere between the artificial and a new kind of natural, the dead and a new kind of living" having "presence as 'local intelligence.'" This property of being alive is significant because it means code can make things do work in the world in an autonomous fashion—that is, it can receive capta and process information, evaluate situations, make decisions, and, most significant, act without human oversight or authorization. When software executes itself, it possesses what Mackenzie (2006) terms *secondary agency*. However, because software is embedded into objects and systems in often subtle and opaque ways, it largely forms a technological unconscious that is noticed only when it performs incorrectly or fails (Thrift 2004b, Graham and Thrift 2007). As a consequence, software often appears to be "automagical" in nature in that it works in ways that are not clear and visible, and it produces complex outcomes that are not easily accounted for by people's everyday experience.

The things that software directs are themselves extremely diverse, varying from simple household items to complex machines and large systems that can work across multiple scales, from the local to the global. In some cases, software augments the use of existing, formerly "dumb," electromechanical technologies such as washing machines and elevators; in other cases, it enables new technological systems to be developed, such as office computing, the Internet, video games, cell phones, and global positioning systems. We see software as embedded in everyday life at four levels of activity, producing what we term *coded objects, coded infrastructures, coded processes*, and *coded assemblages*.

Coded objects are objects that are reliant on software to perform as designed. As we discuss in chapter 3, such objects can be divided into several different classes. Coded machine-readable objects might not have any software embedded in them but rely on

external code to function; DVDs and credit cards are examples. Unless they are worked on by software, they remain inert pieces of plastic, unable to provide entertainment or conduct financial transactions. Other coded objects are dependent on the software embedded within them to perform. Here, software enhances the functional capacity of what were previously dumb objects, such as an electronic scale, or enables objects to be plugged into distributed networks, such as networked vending machines, or underpins the invention of entirely new classes of digital objects, some of which have an awareness of themselves and their relations with the world and record aspects of those relations for future use (examples are MP3 players and mobile devices) (see chapters 3 and 8).

Coded infrastructures are both networks that link coded objects together and infrastructures that are monitored and regulated, fully or in part, by software. Such coded infrastructure includes distributed infrastructures, such as computing networks, communication and broadcast entertainment networks (mail, telephone, cell phones, television, radio, satellite), utility networks (water, electricity, gas, sewer), transport and logistics networks (air, train, road, container shipping), financial networks (bank intranets, electronic fund transfer systems, stock markets), security and policing networks (criminal identification captabases, surveillance cameras), and relatively small-scale and closed systems such as localized monitoring (say, fire and access control alarms and HVAC performance within one building complex), and small but complex systems such as an individual automobile. The geographical extent of distributed infrastructures varies from the global, as with satellite-based global positioning systems (which literally can be accessed from any point on the planet), to more localized coverage, such as a network of traffic lights in a city center.

Coded processes consist of the transactions and flows of digital capta across coded infrastructure. Here, the traffic is more than rudimentary instructions to regulate coded objects within an infrastructure; rather, the flows are structured capta and processed information. Such flows become particularly important when they involve the accessing, updating, and monitoring of relational captabases that hold individual and institutional records that change over time. Such captabases can be accessed at a distance and used to verify, monitor, and regulate user access to a network, update personal files, and sanction a monetary payment, for example. An example of a coded process is the use of an ATM. Here, capta in terms of transaction events are transferred across the coded infrastructure of the bank's secure intranet based on access using a coded object (the customer's bank card), verifying the customer based on a personal identification number (PIN), determining whether a transaction will take place, instructing the ATM to complete an action, and updating the user's bank account. Part of the power of relational captabases is that they hold common fields that allow several captabases to be cross-referenced and compared precisely by software. Other coded processes center on captabases relating individuals and households to bank

accounts, credit cards, mortgages, taxation, insurance, medical treatments, utility use, service contracts, and so on, all of which can be accessed across open or, more commonly, closed networks. Although coded processes are largely invisible and distant, they are revealed to individuals through the fields on official form letters, statements, bills, receipts, printouts, licenses, and so on, and through unique personal identification numbers on the coded objects used to access them (bank and credit cards, library cards, transportation cards, store loyalty cards) and increasing requirements to use passwords. Many of these processes relate to everyday consumption practices and are discussed in chapter 9.

Coded assemblages occur where several different coded infrastructures converge, working together—in nested systems or in parallel, some using coded processes and others not—and become integral to one another over time in producing particular environments, such as automated warehouses, hospitals, transport systems, and supermarkets. For example, the combined coded infrastructures and coded processes of billing, ticketing, check-in, baggage routing, security screening, customs, immigration, air traffic control, airplane instruments, and so on work together to create a coded assemblage that defines and produces airports and passenger air travel (see chapter 7). Similarly, the coded infrastructures of water, electricity, gas, banks and insurers, commodities, Internet, telephone, mail, television, government captabase systems, and so on work in complex choreographies to create an assemblage that produces individual households (see chapter 8). These assemblages are much greater than the sum of their parts, with their interconnection and interdependence enabling the creation of highly complex systems with high utility, efficiency, and productivity.

Computation

That such coded objects, infrastructures, processes, and assemblages exist widely and do work in the world is itself a function of the rapid advances in hardware and the exponential growth in digital computation at increasingly reduced costs, along with the ability to access such computation at a distance through reliable communication technologies. Although the focus of this book is not computer and communications technologies per se, it is important to acknowledge the extent to which computing power has multiplied dramatically in terms of operating speed since the first modern computers were built during World War II, enabling the widespread distribution of software-enabled devices. It is estimated that over the past hundred years, there has been a 1,000,000,000,000,000-fold fall in the cost of computation (Computing Research Association 2003), most of which has occurred in the past fifty years. Nordhaus (2002) calculates that there was approximately a 50 percent increase in computational power each year between 1940 and 2001. In addition, as new electronic and

solid-state hardware technologies for processing have been developed, the cost per million units of computation declined steeply during this period (see figure 1.2).

Computer memory and storage have grown significantly, in tandem with the tremendous improvements in processing power. Gilheany (2000), for example, estimated that since the introduction of the first commercial magnetic disk by IBM in 1956, the cost of storage per gigabyte has fallen by a factor of 1 million to 1. The growth in storage density, as measured in bits per inch on magnetic disks, has even outpaced the upward curve of Moore's law (that the number of transistors on a CPU doubles every two years) and shows few signs of slowing in the near future. The physical space required for data storage has also shrunk dramatically as hard drives and flash memory have become smaller and the density of packing has increased. This growth in storage capabilities enables radically different strategies of information management: deletion of old information is becoming unnecessary, continuous recording is a possibility, and individuals can carry with them enormous amounts of capta in a tiny gadget (see chapter 5).

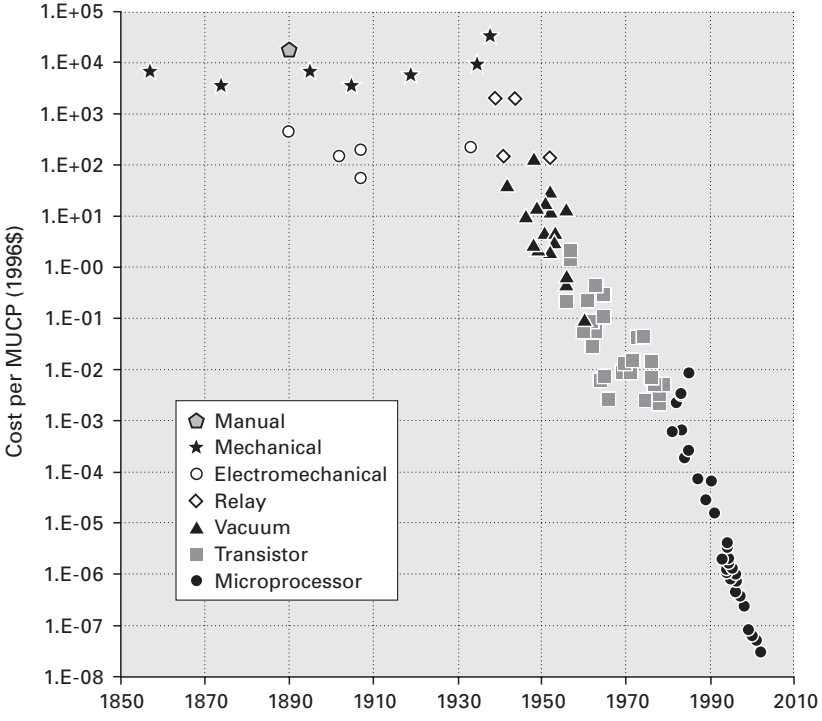


Figure 1.2
Increasing cost efficiencies of computation with a marked step change from mechanical to electronic processing technologies. (Redrawn from Nordhaus 2002, 43)

Communication among computational devices has also become easier, faster, and more widely available. The capability to network devices facilitates all manner of social and economic interactions and transactions and has been underpinned by the rapid development of the Internet over the past two decades. According to George Gilder's "law of telecosm," the "world's total supply of bandwidth will double roughly every four months—or more than four times faster than the rate of advances in computer horsepower [Moore's law]" (Rivlin 2002). In other words, network capacity is growing faster than demand even with increasingly information-rich applications. Network bandwidth is also becoming progressively more diffused geographically at much lower costs (notwithstanding the ongoing concerns over digital divides and the unevenness of telecommunication pricing and regulation). Many people now expect continuous network access regardless of where they are.

Yet it is not just the raw numbers in terms of CPU clock speeds, gigabytes of disk space, and download speed that matter. Pragmatic issues such as physical design, interface usability, reliability, and real cost for daily use have undergone significant improvements that have made computation attractive in terms of widespread consumer confidence and affordability. As a result, social dispositions toward software-enabled technologies have become favorable, resulting in hundreds of millions of computational devices being distributed, embedded into, and carried around environments; these devices often bear little resemblance to desktop computers, and the work that many of them do is hidden from view. As we discuss in detail in chapter 10, some commentators say that we are entering a new age—Greenfield (2006) refers to this as *everyware*—in which computing becomes pervasive and ubiquitous. In this new era, software mediates almost every aspect of everyday life.

The Power of Code

Taken together, coded objects, infrastructures, processes, and assemblages mediate, supplement, augment, monitor, regulate, facilitate, and ultimately produce collective life. They actively shape people's daily interactions and transactions, and mediate all manner of practices in entertainment, communication, and mobilities. As we explore in chapter 2, software has the power to shape the world in a number of ways. It has dramatically increased the capacity of both people and institutions to process information in terms of volume, speed of processing, and the complexity of operations, and at a very low cost per transaction. It has enabled forms of automation, the monitoring and controlling of systems from a distance, the reconfiguring and rejuvenation of established industries, the development of new forms of labor practices and paid work, the reorganization and recombination of social and economic formations at different scales, and it has produced many innovations. And because software can be programmed to read inputs to a system, and evaluate and react to those assessments, it

has a significant degree of autonomy. Consequently, as we argue in chapter 5, most people in Western nations are living in a machine-readable and coded world—that is, a world where information is routinely collected, processed, and acted on by software without human intervention.

In many cases, the power of software is significant but banal—a digital alarm clock that wakes a worker or the ATM that provides her with access to money when banks are not open for business. Here, if the software crashes, then its consequence is typically frustration and localized inconvenience. In other cases, software is the difference between something happening or not, because manual systems have been entirely replaced by digital systems. And when some software systems crash, they can create major incidents with serious economic and political effects, and even life-threatening situations. The crash of the Tokyo Air Traffic Control Center in March 2003 meant the cancellation of over 203 flights (Risks List 2003), and seemingly minor failures in routine monitoring software systems at FirstEnergy in Akron, Ohio, were key contributing factors in the large-scale power outage affecting millions of people in the U.S. Northeast in 2003 (U.S.-Canada Power System Outage Task Force 2004). A great deal of resources are expended to keep digital systems that rely on software in working order; much of this routine maintenance and repair is hidden labor but is nonetheless vital to the information society (Graham and Thrift 2007).

Perhaps the best illustration of the contemporary social and economic importance of software was the widespread concern at the end of 1990s associated with the Y2K millennium bug, which triggered a wholesale overhaul of software systems in many nations. The cost to the U.S. federal government alone was estimated at $8.34 billion, and governments and businesses across the world spent an estimated $200 billion to $600 billion to address the problem (Bennett and Dodd 2000). Such investment, and media hype and speculation, would not have been expended if there had not been genuine worry that services in the public and private sectors would suffer serious disruption and possible collapse. Indeed, such is the reliance by governments and businesses on a raft of office applications and larger software systems that it is now unthinkable to backtrack to a predigital age: the nature of tasks has changed, staff levels have been reduced and deskilled in many cases, and operational networks and transactions have become much more complex and interdependent.

Significantly, software engenders both forces of empowerment and discipline, opportunities and threats. Software is enabling the realization of many new forms of creative technology and novel kinds of art, play, and recreation; it makes social and economic processes more efficient, effective, and productive; and it creates new opportunities and markets. At the same time, software has underpinned the development of a broad range of technologies that more efficiently and successfully represent, collate, sort, categorize, match, profile, and regulate people, processes, and places.

Software is at the heart of new modes of invasive and dynamic surveillance and the creation of systems that never forget (see chapters 5, 10, and 11).

Social analysts have a tendency to focus on the active role of software in regulatory technologies, in processing and analyzing capta about people, and in systems of social control. From this perspective, it is difficult not to become pessimistic about the work software does in the world—its use to determine, discipline, and potentially discriminate. And yet the reason that digital technologies are so popular is that they make societies safer, healthier, and richer overall even as they do the work to regulate societies. (We acknowledge that the beneficial outcomes are not necessarily equitably distributed.) Software development has provided innovations across many fields; led to new job opportunities in cleaner industries; driven fresh rounds of capital investment in new and old business sectors; provided more and wider entertainment and retail choices; automated everyday tasks; increased access to credit; opened up new forms of social communication and driven down their cost; enabled new forms of creativity, knowledge production, and artistic practice; opened up original ludic possibilities and ways of recording personal experiences and memories; led to new media for political organization and oppositional activities; and so on (see chapter 6). In this sense, a key aspect of the power of software lies in how it seduces. In Althusser's (1971) terms, software-driven technologies induce a process of interpellation, wherein people willingly and voluntarily subscribe to and desire their logic, trading potential disciplinary effects against benefits gained. And the benefits are often substantial and, in a very quotidian sense, irresistible. Perhaps rather than trying to determine whether the work software does is good or bad, it is better to see it as productive in the broad sense—it makes things happen. We need to understand how this production unfolds in different social and spatial contexts.

Software, Society, and Space

Until recently software was largely ignored by the social sciences and humanities. Instead, with perhaps the exception of research in computer-mediated communication and computer-supported cooperative work, scholars and commentators tended to focus more broadly on the information and communication technologies (ICTs) that software enables, in particular the Internet, rather than to more specifically consider the role of code in relation to those technologies and wider society. This has led over the past fifteen years or so to a burgeoning set of studies focusing on what Castells (1996) has called the *network society*. The general thesis is that ICTs are transformative technologies that enable a shift from an industrial to a postindustrial society by altering the conditions through which social and economic relations take place. ICTs are reconfiguring the means by which capital is generated by allowing businesses to reorganize their operations advantageously, change working practices, reduce costs,

increase productivity, and diversify into new products and markets (Castells 1996; Kitchin 1998). Here, capta generation, processing, and information exchange are key to developing knowledge and extracting value. Similarly, social relations are speeded up and altered through new forms of communication media such as e-mail, Web pages, virtual worlds, chatrooms, and mobile phones that allow experimentation with identity and novel social networks to be developed (Rheingold 1993; Turkle 1995; Wellman and Haythornthwaite 2002).

As analysts such as Foth (2008), Mitchell (1995), Graham and Marvin (1996, 2001), and Townsend (2003) detail, ICTs are also having material effects on how cities and regions are configured, built, and managed with the development of smart buildings, the networking of physical infrastructure, the use of traffic management and other information and control systems, and so on. This is what Batty (1997, 155) has termed the *computable city,* noting that "planners . . . are accustomed to using computers to advance our science and art but it would appear that the city itself is turning into a constellation of computers." Such a city is perhaps best illustrated by the proliferation of windowless, semisecret, and hermetically sealed control rooms, with their banks of screens showing software-generated real-time inscriptions of urban infrastructures and flows (figure 1.3).

Software studies is a fledgling field. Although work within this field predates the new millennium, the first notion of the field itself can be traced to Manovich (2000, 48), who argued that "to understand the logic of new media we need to turn to computer science. It is there that we may expect to find the new terms, categories and operations which characterize media which became programmable. From media studies, we move to something which can be called software studies; from media theory—to software theory." Complementing the work of computer scientists on the mechanics of software development and human computer interaction, and research on digital technologies more generally, social theorists, media critics, and artists have begun to study the social politics of software: how it is written and developed; how software does work in the world to produce new subjects, practices, mobilities, transactions, and interactions; the nature of the software industry; and the social, economic, political, and cultural consequences of code on different domains, such as business, health, education, and entertainment. Manovich (2008, 6) asserts, "I think that software studies has to investigate both the role of software in forming contemporary culture, and cultural, social, and economic forces that are shaping development of software itself." In conjunction, Fuller (2008, 2) argues that the field "proposes that software can be seen as an object of study and an area of practice for the kinds of thinking and areas of work that have not historically 'owned' software, or indeed often had much of use to say about it."

The difference between software studies and those more broadly studying the digital technologies they enable could be characterized as the difference between

studying the underlying epidemiology of ill health and the effects of ill health on the world. While one can say a great deal about the relationship between health and society by studying broadly how ill health affects social relations, one can gain further insight by considering the specifics of different diseases, their etiology (causes, origins, evolution, and implications), and how these manifest themselves in shaping social relations.

Software studies focuses on the etiology of code and how code makes digital technologies what they are and shapes what they do. It seeks to open the black box of processors and arcane algorithms to understand how software—its lines and routines of code—does work in the world by instructing various technologies how to act. Important formative works include Galloway's *Protocol* (2004); Fuller's *Behind the Blip* (2003), *Media Ecologies* (2005), and *Software Studies: A Lexicon* (2008); Lessig's *Code and Other Laws of Cyberspace* (1999); Manovich's *The Language of New Media* (2000) and *Software Takes Command* (2008); Hayles's *My Mother Was a Computer* (2005); and Mackenzie's *Cutting Code* (2006).

These studies demonstrate that software is a social-material production with a profound influence on everyday life. All too often, however, they focus on the role of software in social formation, organization, and regulation, as if people and things exist in time only, with space a mere neutral backdrop. What this produces is historically nuanced but largely aspatial accounts of the relationship of software, technology, and society. As geographers and others argue, however, people and things do not operate independent of space. Space is not simply a container in which things happen; rather, spaces are subtly evolving layers of context and practices that fold together people and things and actively shape social relations. Software and the work it does are the products of people and things in time and space, and it has consequences for people and things in time and space. Software is thus bound up in, and contributes to, complex discursive and material practices, relating to both living and nonliving, which work across geographic scales and times to produce complex spatialities. From this perspective, society, space, and time are co-constitutive—processes that are at once social, spatial, and temporal in nature and produce diverse spatialities. Software matters because it alters the conditions through which society, space, and time, and thus spatiality, are produced.

Our principal argument, then, is that an analysis of software requires a thoroughly spatial approach. To date, however, geographers and spatial theorists, like other social scientists, have tended to concentrate their analysis on the technologies that software enables and their effects in the world. This work has examined the effects of ICTs on time-space convergence (acceleration in the time taken to travel or communicate between places) and distanciation (control from a distance); the spatial economics of business and patterns of capital investment and innovations across spatial scales; the ways in which cities and regions are being reconfigured and restructured; and notions

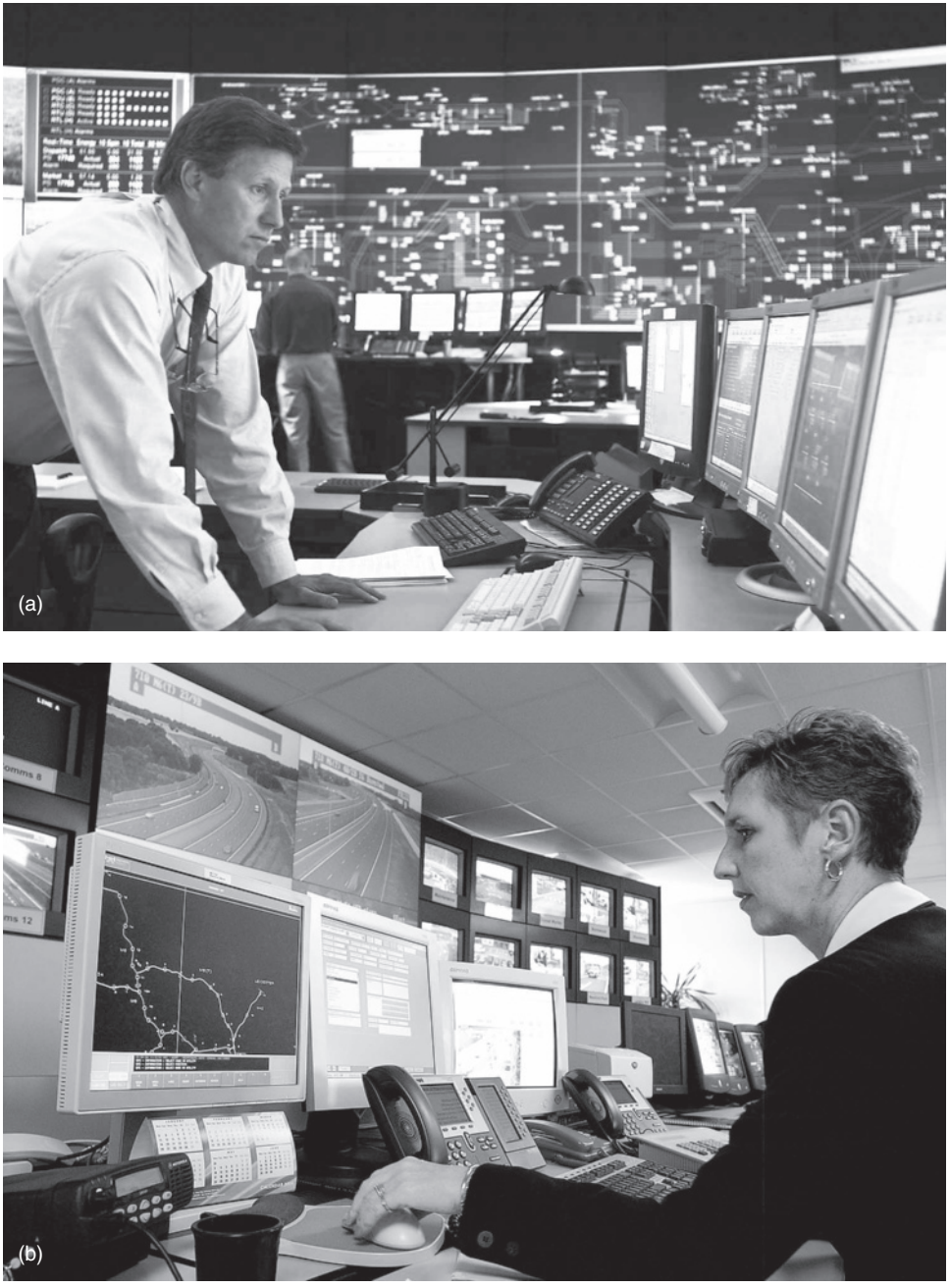


Figure 1.3
Control rooms monitoring the city through code. (*a*) Electricity supply (*Source*: courtesy of Independent Electricity System Operator, Ontario, www.ieso.ca). (*b*) Road traffic (*Source*: courtesy of Midland Expressway Ltd., www.m6toll.co.uk). (*c*) Video surveillance and security for a shopping center (*Source*: courtesy of Arndale, www.manchesterarndale.com). (*d*) Water resources (*Source*: courtesy of Barco, www.barco.com).

Figure 1.3
(continued)

of place, identity, and spatially grounded identities (see Daniels et al. 2006; Dodge and Kitchin 2000; Graham and Marvin 2001; Wheeler, Aoyama, and Warf 2000; Wilson and Corey 2000; Zook 2005). Although there is much to learn from spatial accounts of the nature and effects of ICTs on various sociospatial domains and at different scales, it is imperative, we argue, for spatial theorists to think more specifically about how software underpins the nature of ICT and shapes its functioning and effects, and more broadly about the work software does in the world through the growing array of technologies used in everyday situations.

In this book, we seek to detail and theorize the ways in which software creates new spatialities of everyday life and new modes of governance and creativity (which are themselves inherently spatial), and we provide a set of conceptual tools for analyzing its nature and consequences. In so doing, and outlined more fully in chapters 4 and 5, we develop a distinct understanding of spatiality that conceives the world as ontogenetic in formulation (that is, constantly in a state of becoming) and rethink software-based governance as a system of automated management. Our analysis does not stand alone; it seeks to complement and extend a small but significant line of work by geographers and allied scholars on the task of describing and explaining the geographies of software (Adey 2004; Budd and Adey 2009; Crang and Graham 2007; Graham 2005; Mitchell 2004; McCullough 2004; Thrift and French 2002; Zook and Graham 2007).

Software, we argue, alternatively modulates how space comes into being through a process of transduction (the constant making anew of a domain in reiterative and transformative practices). Space from this perspective is an event or a doing—a set of unfolding practices that lack a secure ontology—rather than a container or a plane or a predetermined social production that is ontologically fixed. In turn, society consists of collectives that are hybrid assemblages of humans and many kinds of nonhumans (Latour 1993), wherein the relationship between people, material technology, time, and space is contingent, relational, productive, and dynamic. Taking the ideas of transduction and automated management together, our central argument is that the spatialities and governance of everyday life unfold in diverse ways through the mutual constitution of software and sociospatial practices. The nature of this mutual constitution is captured in our concept of code/space.

What is Code/Space?

Code/space occurs when software and the spatiality of everyday life become mutually constituted, that is, produced through one another. Here, spatiality is the product of code, and the code exists primarily in order to produce a particular spatiality. In other words, a dyadic relationship exists between code and spatiality. For example, a check-in

area at an airport can be described as a code/space. The spatiality of the check-in area is dependent on software. If the software crashes, the area reverts from a space in which to check in to a fairly chaotic waiting room. There is no other way of checking a person onto a flight because manual procedures have been phased out due to security concerns, so the production of space is dependent on code. Another example is a supermarket checkout. All supermarkets and large stores rely on computerized cash registers to process purchases. If the computer or the information system behind it crashes, shoppers cannot purchase goods, and in a functional sense, the space effectively ceases to be a supermarket instead becoming a temporary warehouse until such time as the code becomes (re)activated. The facilities to process payments manually have been discontinued, staff are not trained to process goods manually (they no longer rote-learn the price of goods), and prices are not usually printed on items. In other words, the sociospatial production of the supermarket is functionally dependent on code.

People regularly coproduce code/spaces, even if they are not always aware they are doing so, and as we demonstrate throughout this book, they are increasingly common in a range of everyday contexts. Any space that is dependent on software-driven technologies to function as intended constitutes a code/space: workplaces dependent on office applications such as word processing, spreadsheets, shared calendars, information systems, networked printers, e-mail, and intranets; aspects of the urban environment reliant on building and infrastructural management systems; many forms of transport, including nearly all aspects of air travel and substantial portions of road and rail travel; and large components of the communications, media, finance, and entertainment industries. Many of the rooms that people live in; the offices, shops, and factories they work in; and the vehicles they travel in are code/spaces. It is little wonder that many commentators are speculating that most people in Western society are entering an age of "everyware" (Greenfield 2006, see also chapter 10).

Given that many of these code/spaces are the product of coded infrastructure, their production is stretched out across extended network architectures, making them simultaneously local and global, grounded by spatiality in certain locations, but accessible from anywhere across the network, and linked together into chains that stretch across space and time to connect start and end nodes into complex webs of interactions and transactions. Any space that has the latent capacity to be transduced by code constitutes a code/space at the moment of that transduction. So, for example, spaces that have wireless access to computation and communication are transduced by a mobile device accessing that network; for example, the laptop computer accessing a wireless network transduces the café, the train station, the park bench, and so on into a work space for that person. Code/space is thus both territorialized (in the case of a supermarket) and deterritorialized (in the case of mobile transductions). The transduction of code/spaces then often lacks singular, easily identifiable points of

control or measurable extents, and they have a complexity much greater than the sum of their parts.

Of course, not all social interactions with software transduce code/spaces. Although much spatiality is dependent on software, software merely augments its transduction in other cases. We term such cases *coded spaces*—spaces where software makes a difference to the transduction of spatiality but the relationship between code and space is not mutually constituted. For example, a presentation to an audience using PowerPoint slides might be considered a coded space. The digital projection of the slides makes a difference to the spatiality of the lecture theater, influencing the performance of the speaker and the ability of the audience to understand the talk. However, if the computer crashes, the speaker can still deliver the rest of the lecture, but perhaps not as efficiently or effectively as when the software worked.

In other words, the distinction between coded space and code/space is not a matter of the amount of code (in terms of the number of lines of code or the density of software systems). Rather a code/space is dependent on the dyadic relationship between code and space. This relationship is so all embracing that if half of the dyad is put out of action, the intended code/space is not produced: the check-in area at the airport does not facilitate travel; the store does not operate as a store. Here, "software quite literally conditions . . . existence" (Thrift and French 2002, 312). In coded space, software matters to the production and functioning of a space, but if the code fails, the space continues to function as intended, although not necessarily as efficiently or cost efficiently, or safely. Here, the role of code is often one of augmentation, facilitation, monitoring, and so on rather than control and regulation.

As we detail in depth in chapter 4, it is important to note that the relationship between software and space is neither deterministic (that is, code determines in absolute, nonnegotiable means the production of space and the sociospatial interactions that occur within them) nor universal (that such determinations occur in all such spaces and at all times in a simple cause-and-effect manner). Rather, how code/space emerges through practice is contingent, relational, and context dependent. Code/space unfolds in multifarious and imperfect ways, embodied through the performance and often unpredictable interactions of the people within the space (between people and between people and code). Code/space is thus inconsistently transduced; it is never manufactured and experienced in the same way.

Discursive Regimes Underpinning Code/Space

The adoption of software and digital technologies, and the systems, networks, and ways of doing they underpin, have been complemented by a broad set of discursive regimes that have sought to justify their development and naturalize their use. For Foucault (1977), a discursive regime is a set of interlocking discourses that sustain and

reproduce, through processes of definition and exclusion, intelligibility and legitimacy, a particular set of sociospatial conditions. Such a regime provides the rationale for how sociospatial relations are predominantly produced, legitimating the use of discursive and material practices that shape their production.

As we discuss in chapter 5 and illustrate in subsequent chapters on travel, home, and consumption, the development and employment of different types of software and digital technologies are underpinned by their own particular, distinctive discursive regime. That said, they usually consist of an amalgam of a number of common discourses: safety, security, efficiency, antifraud, empowerment, productivity, reliability, flexibility, economic rationality, and competitive advantage. In other words, they argue that the deployment of software will improve the safety of individuals and society more broadly; make society and travel more secure; make government or business more efficient; make the fight against crime more effective; empower people to be more creative and innovative; and so on. These discourses are often promoted by government in tandem with business, driven by the interests of capitalism and, increasingly, the agenda of neoliberalism focused on the delivery of social services for profit within a target-driven culture.

The constituent elements of a discursive regime work to promote and make commonsense their message, but also to condition and discipline. Their power is persuading people to their logic—to believe and act in relation to this logic. As Foucault (1977, 1978) noted, however, a discursive regime does not operate solely from the top downward, but through diffused microcircuits of power, the outcome of processes of regulation, self-regulation, and localized resistance. As such, people are not simply passive subjects, disciplined and interpellated in linear and unproblematic ways by discursive regimes. Rather, as with the technologies themselves, such discourses are open to rupture: subversion, denial, and transgression by flourishing software hacking communities, anticorporate web sites, online activist networks, legal challenges to security and surveillance, and campaigns concerning privacy and confidentiality, for example. In this sense, power is not captive, purely in the hands of an unseen elite, although the discursive regime operates—in conjunction with the operation of code/space—to try to (re)produce such a hegemonic order. As such, code/spaces and their discursive regimes work to reinforce and deepen their logic and reproduction, at the same time as others seek to undermine, resist, and transform their hegemonic status. Software opens up new spaces as much as it closes existing ones. Accordingly, Amin and Thrift (2002, 128) argue that because "the networks of control that snake their way through cities are necessarily oligoptic, not panoptic: they do not fit together. They will produce various spaces and times, but they cannot fill out the whole space of the city—in part because they cannot reach everywhere, in part because they therefore cannot know all spaces and times, and in part because many new spaces and times remain to be invented."

Interestingly, given the increasing power and role of software, resistance to digital technologies has been remarkably mute despite widespread cynicism over the perceived negative effects of computerization. Thrift and French (2002, 313) note, "Even though software has infused into the very fabric of everyday life—just like the automobile—it brings no such level of questioning in its wake." There seem to be a number of reasons for this: the majority of people have been persuaded to its utility and how it is made rational and natural by discursive regimes; people are empowered and recognize the benefits of software technologies to their everyday lives; people see the changes that are occurring as simply an extension of previous systems to which they are already conditioned; how software is incrementally employed is seen as an inherent aspect of how things are now done and are therefore unchallengeable; the employment of software is seen as largely benign and routine rather than threatening and invasive; and people are worried by the consequences of protest (e.g., denial of services or mistreatment) and so refrain from doing so. Whatever the reason, there has been little scrutiny of the extent to which software has become embedded into everyday life or of the discourses that underpin, and subtly and explicitly promote, their adoption.

The Book

Code/Space is principally a book about the relationship of software, space, and society. Its main focus concerns how software, in its many forms, enables the production of coded objects, infrastructures, processes, and assemblages that do work in the world and produce the code/spaces and coded spaces that increasingly constitute the spatialities of everyday life. The goal "is not therefore to stage some revelation of a supposed hidden truth of software, to unmask its esoteric reality, but to see what it is and what it can be coupled with: a rich seam of paradoxical conjunctions in which the speed and rationality of computation meets its ostensible outside" (Piet Zwart Institute 2006).

In the following chapter, we discuss the nature of software and how it is both a product of the world and a producer of the world. In part II, we theorize how and why software makes a difference to society, providing a set of conceptual ideas and tools to think through how code transforms the nature of objects and infrastructures, transduces space, transforms modes of governmentality and governance, and engenders new forms of creativity and empowerment. In part III, we then employ these concepts to analyze the diverse ways in which the employment of software shapes the spatialities of everyday life, focusing on aspects of travel, home, and consumption. In each of these domains, social activities are now regularly transduced as code/spaces. In part IV, we examine likely future code/spaces and the drive toward everyware, explore the ethical dilemmas of such potentialities, and think about how code/spaces

should be researched as the field of software studies develops. Our conclusions offer a provisional manifesto for critical scholarship into code—a new kind of social science focused on explaining the social, economic, and spatial contours of software. The glossary provides succinct definitions of key terms, especially technical terms and neologisms that may not be familiar to some readers.

Software matters in ways that extend well beyond simple functional or instrumental utility. *Code/Space* explains and demonstrates why it matters.

2 The Nature of Software

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

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 veri-

fied, 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;
```

```
                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.

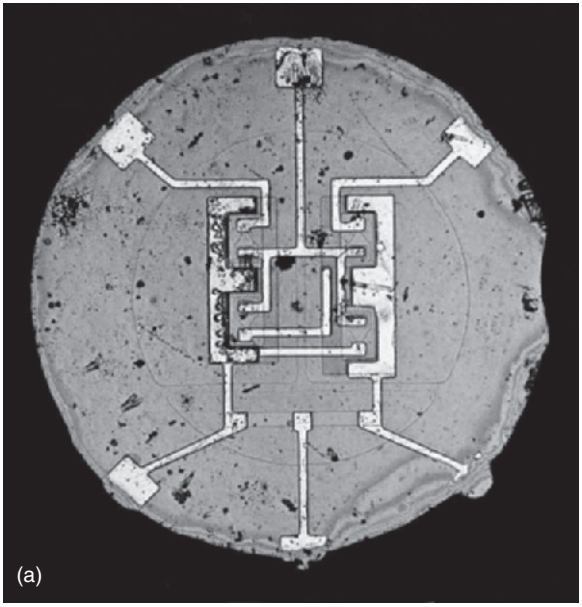
Regardless of the nature of programming, the code created is the manifestation of a system of thought—an expression of how the world can be captured, represented, processed, and modeled computationally with the outcome subsequently doing work in the world. Programming then fundamentally seeks to capture 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

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



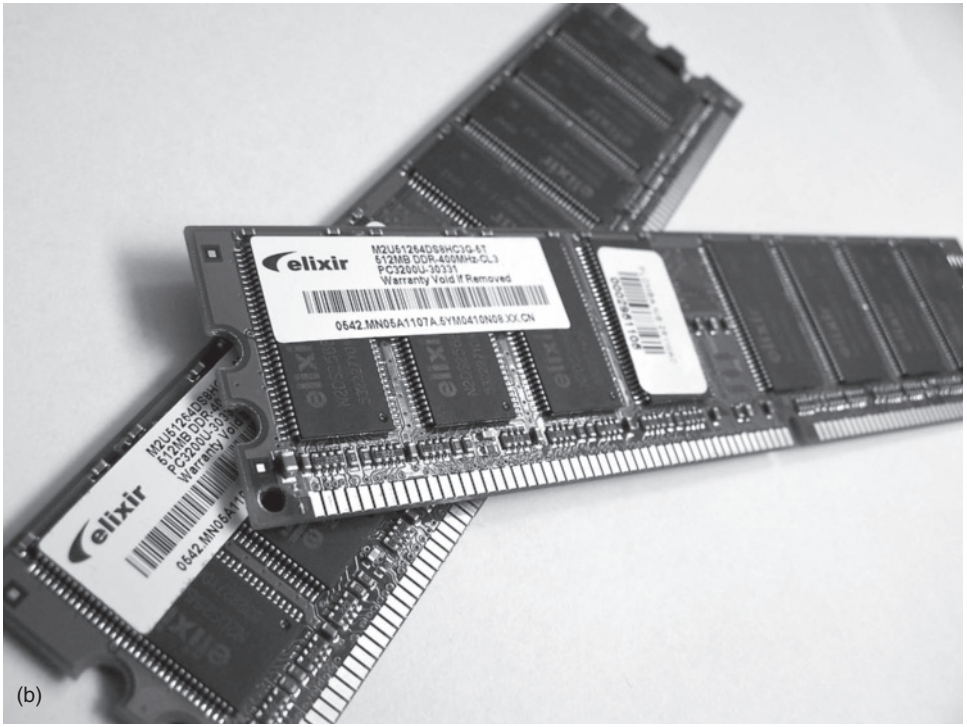


Figure 2.1
Computation microspaces where software works and capta dwells. (*a*) Silicon logic gate (*Source*: www.technologyreview.com/article/21886/). (*b*) Memory cards (*Source*: http://images.suite101.com/538579_com_ram.jpg). (*c*) Circuit board (*Source*: www.technologyreview.com/article/21886/). (*d*) Hard drive (*Source*: http://commons.wikimedia.org/wiki/File:Hard_disk.jpg).

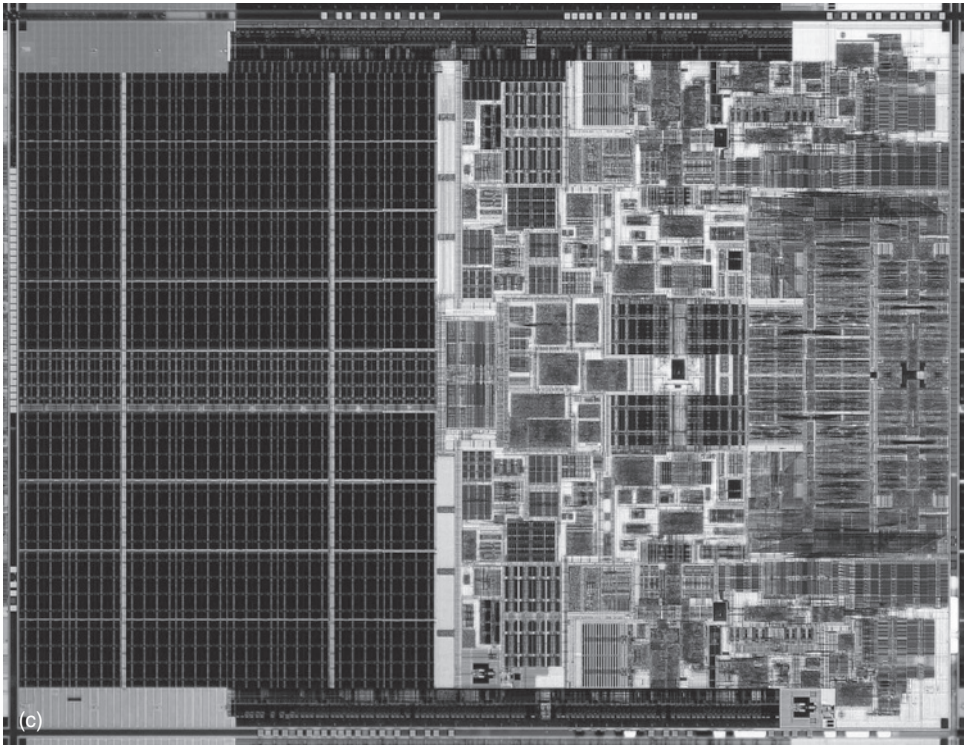


Figure 2.1
(continued)

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).

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