



# *Introduction to Computer Technology, Network Economics, and Intellectual Property Law*

Computer software and Internet commerce are among the fastest growing and most promising industries in the United States. A recent government report notes that more than half of U.S. nonfarm industries either produce information technology (IT) directly or invest in and use information technology products and services. U.S. Commerce Department, *The Emerging Digital Economy II* (1999). The information technology sector of the U.S. economy represented 8 percent of gross domestic product (GDP) in 1999, accounting for more than \$700 billion. Computer software accounted for \$200 billion of this total. The IT sector of the U.S. economy has steadily increased its share of the GDP in the 1990s and shows no sign of slowing down. These patterns can be seen throughout the global economy. *A World Gone Soft: A Survey of the Software Industry*, *The Economist* (May 25, 1996).

While firms such as Intel, Microsoft, Compaq, IBM, Cisco, AOL, and Amazon.com attract much of the attention in the IT marketplace, the IT industries touch almost all aspects of the modern economy. For example, traditional manufacturing firms, such as General Motors, make significant use of computers, computer software, and computer networks in their businesses. Automobile manufacturers use CAD (computer-aided design) software to design new vehicles, CAM (computer-aided manufacturing) software to assemble these designs, and digital networks to purchase component parts and to distribute vehicles to customers. Few businesses, government agencies, schools, or other organizations operate today without extensive use of computer technology and digital networks. An increasingly wide array of companies—whether they sell information, cars, or anything else—use digital networks, principally the Internet, to market products and transact business. While it is easy to scoff at estimates of the potential growth of global electronic commerce because they seem like (and probably are) rank guesswork, electronic commerce

has surpassed what once seemed like exaggerated estimates. The Emerging Digital Economy II report notes that in 1997 “private analysts forecast that the value of Internet retailing could reach \$7 billion by 2000—a level surpassed by nearly 50 per cent in 1998.” While the popular press has mainly concentrated on the growth of Internet business-to-consumer companies, such as Amazon.com, many industry experts believe that business-to-business ecommerce will be larger and have more far-reaching implications for the U.S. economy. Internet technology makes possible great efficiencies in the ways businesses are structured, distribute product and service information, and conduct transactions. The extent of these possibilities are just beginning to emerge.

This chapter describes the early history of computing in Section A. It explains how computers work in very basic terms in Section B, and introduces the principal models of software engineering in Section C. Section D discusses the distinctive economics of computer software and network markets. Finally, Section E offers an overview of the principal intellectual property laws protecting computer technology. Subsequent chapters address these legal regimes in greater detail.

Readers with a strong understanding of how computers work and how programs are written should feel comfortable in skipping or skimming Sections A through C. They should read the balance of the chapter in detail, however, because the economics of computer software and the intellectual property background may be unfamiliar—and will be important in the chapters to come.

Students who do not have a strong background in computer technology should review Sections A through C at this time. For those students, a brief note on methodology in this chapter is in order. Our goal in providing information about the computer industry is to offer students essential background on how computer software works and how markets for computer software function. We do not intend this introduction to provide a complete understanding of the field. In our summaries below, we reference a number of excellent sources providing detailed background. The reader who is interested in more detail than we can possibly provide here should seek out these sources.

## **A. THE EARLY HISTORY OF COMPUTERS**

This Section introduces the reader to the early history of computer hardware and software. The purpose of this Section is to describe the enormous changes that occurred in the early days of the computer industry in order to provide context for the discussions that will follow. This Section does not describe events up to the present day. More recent developments (including the growth of the Internet) are discussed in the sections that follow, and in later chapters, where modern technological developments often present new legal issues.

Following the invention of the abacus approximately 5,000 years ago, the field of computing machines did not develop significantly until the eighteenth century. Leonardo da Vinci (1425-1519) sketched some designs for mechanical adding machines. Blaise Pascal (1623-1662) invented and built the “Pascaline,” a sophisticated mechanical device for counting. Although not commercially successful because of its cost and delicate construction, the counting-wheel design served as the basis for most mechanical calculators until the 1960s. At the turn of the nineteenth century, Joseph-Marie Jaquard (1752-1834) introduced a new loom technology that used punched cards to control the movement of needles, thread, and fabric to create distinctive patterns through a binary mechanical automation technology. In the mid-nineteenth century, Charles Babbage envisioned mechanical devices (the Difference Engine and the Analytical Engine) to perform arithmetic operations. His designs, involving thousands of gears, proved impractical. One of his students, Lady Ada August Lovelace, proposed the use of punched cards to automate the operation of such devices.

Toward the end of the nineteenth century, a U.S. Census Bureau agent named Herman Hollerith developed a punched-card tabulating machine to automate the census. Drawing on the use of “punched photography” by railroads (to encrypt passengers’ hair and eye color on tickets), Hollerith proposed the encoding of census data for each person on a separate card that could be tabulated mechanically. After developing this technology for the Census Bureau, he formed the Tabulating Machine Company in 1896 to serve the growing demand for office machinery, such as typewriters, record-keeping systems, and adding machines. The company grew through the expansion of its business and merger with other office supply companies, and in 1924 Thomas J. Watson, the company’s general manager, changed the company’s name to International Business Machines Corporation (IBM). By the late 1920s, IBM was the fourth largest office machine supplier in the world, behind Remington-Rand, National Cash Register (NCR), and Burroughs Adding Machine Company. IBM made numerous improvements to tabulating technology during the 1920s and 1930s and eventually developed a machine that could compare cards, a significant innovation that enabled machines to perform simple logic (if—then) operations.

## 1. Computer Hardware

The critical breakthrough defining modern computers was the harnessing of electrical impulses to process information. In 1939, Professor John Vincent Atanasoff, with the help of his graduate student, Clifford Berry, developed the first electronic calculating machine. This computer could solve relatively complicated physics computations. Atanasoff and Berry built a more sophisticated version, the ABC (Atanasoff Berry Computer), in 1942. Shortly thereafter, driven in part by wartime demand for computing technology, Professor Howard Aiken, funded in substantial part by IBM, developed a massive electro-

mechanical computer (MARK I). This machine contained three-fourths of a million parts, hundreds of miles of wire, and was 51 feet long, 8 feet high, and 2 feet deep. It could perform three additions a second and one multiplication every six seconds. Although it used an electric motor and a serial collection of electromechanical calculators, the MARK I was in many respects similar to the design of Babbage's analytical engine.

At about this same time, Dr. John Mauchly persuaded the U.S. Army to fund the development of a new computing device to compute trajectory tables to improve the targeting of ordnance. Mauchly envisioned using vacuum tubes rather than mechanical relays to store binary information. In collaboration with J. Presper Eckert, Jr., a young electrical engineer, Mauchly completed the electronic numerical integrator and computer (ENIAC) in 1946. This computer occupied 15,000 square feet, weighed 30 tons, and contained 18,000 vacuum tubes. It operated in decimal (rather than binary) format and therefore needed 10 vacuum tubes to represent a single digit. The ENIAC could perform over 80 additions or 8 multiplication operations per second.

The flexibility provided by programmability greatly enhanced the utility of computers. In the early 1950s, Mauchly and Eckert developed the first commercially viable electronic computer, the universal automatic computer (UNIVAC I) for Remington-Rand Corporation. Limitations on electronic technology, however, constrained the computing power of the first generation of computers. Vacuum tubes, which were bulky, failed frequently, consumed large amounts of energy, and generated substantial heat. This first generation of computers was programmed in binary code (zeros and ones), which could be understood by only a few specialists. IBM introduced its first commercial computer, the IBM 650, in 1954. IBM made incremental improvements to this technology and emerged as the market leader.

Because computers use binary electronic switches to store and process information, the great challenge for the computer industry was to reduce the size of these switches. The second generation of computers replaced vacuum tubes with transistors, which were smaller, required less power, and ran without generating significant heat. This and other innovations in data storage technology made computers smaller, faster, and more reliable. The first scientific computer using transistors was the IBM 7090. A second important innovation of this era was the development of high-level computer languages, which enabled computer specialists to write programs using coded instructions that resemble human language. The IBM 705, introduced in 1959, used the FORTRAN language processor. This model became the standard machine for large-scale data processing companies. Notwithstanding these innovations, computers of this generation remained complex and expensive because circuits had to be wired by hand.

The development of integrated circuits enabled computer manufacturers to incorporate many transistors within the layers of semiconductor material. The greater computing power and efficiency of computers brought the cost of data processing services within the reach of an increasing number of businesses. Many businesses contracted with companies specializing in data processing

services, and a few acquired their own computers. IBM's 360 series of mainframe computers emerged during this period as the market leader. These machines used a single machine language. As businesses upgraded their equipment within the 360 series, they could continue to use the same computer programs. This increased the benefit of owning a computer (rather than outsourcing data processing) and expanded the mainframe market. This larger market generated greater demand for computer programmers and spawned new companies to provide computer-related services. An independent software industry began to emerge.

The 1960s and 1970s also witnessed the implementation of time-sharing and telecommunication technologies, which enable multiple users to access a computer from remote terminals. In addition, computers developed during this period could handle multiple tasks simultaneously (parallel processing and multiprogramming).

In 1965, the Digital Equipment Corporation (DEC) introduced the first minicomputer, the PDP-8 (programmed data processor). This machine was substantially smaller and about one-fourth the price of mainframe computers. Minicomputers substantially widened the market for computers and computer programmers. Domestic consumers purchased 260 minicomputers and 5,350 mainframes in 1965. Minicomputer unit sales surpassed mainframe unit sales by 1974. By the 1970s, computers incorporated "semiconductor chips" no larger than a human fingernail and containing more than 100,000 transistors. As chip technology advanced, the size of computers decreased while their computing power increased. Semiconductor chips today can hold many millions of transistors. For the past two decades, the memory capacity of a semiconductor chip has doubled approximately every 18 months.

In the early 1970s, Intel Corporation developed the microprocessor, a chip that contains the entire control unit of a computer. Very large scale integration (VLSI) technology led to the development of the microcomputer. Originally oriented toward computer hobbyists, microcomputers came to dominate the computer industry by the mid-1980s. With its Apple II computer system, which included a keyboard, monitor, floppy disk drive, and operating system, Apple Computer vastly expanded the market for computers. Microcomputer unit sales surpassed minicomputer unit sales in 1976, their second year of production. By 1986, sales of microcomputers (costing less than \$1,000) reached approximately 4 million units and produced revenues of almost \$12 billion, giving microcomputers the largest share of computer industry revenues.

The rapid growth of the microcomputer sector of the industry spurred the emergence of independent software vendors (ISVs) who developed mass-marketed programs for this growing market of versatile machines. Microcomputer owners were anxious to experiment with different programs. The cost of developing software for these machines was relatively low, product cycles were short, and there was constant pressure to upgrade products.

IBM entered the microcomputer market in 1981 with its PC (personal computer) product. The IBM PC utilized an Intel microprocessor (16-bit 8088 chip) and an operating system, PC-DOS (disk operating system), licensed from

Microsoft, then a fledgling company. Microsoft's MS-DOS is a single-tasking, single-user operating system with a command-line interface. Like other operating systems, MS-DOS oversees operations such as disk input and output, video support, keyboard control, and many internal functions related to program execution and file maintenance.

IBM's strong trademark in the business computer industry as well as its vast distribution network for computers enabled IBM to rapidly attract customers for its PC product. Many ISVs and hardware manufacturers developed and marketed software and peripheral products to run on the IBM PC. IBM actively encouraged ISVs and the makers of peripheral equipment (e.g., printers and monitors) to develop products for the PC. While promoting an "open architecture" with regard to these sectors of the industry, IBM included a specialized chip (BIOS)<sup>1</sup> for transferring data within the PC that hindered other original equipment manufacturers (OEMs) from offering fully compatible computer systems. This enabled IBM to charge premium prices for its PC product.

The rapid success of the IBM PC spurred ISVs to develop a wide range of programs to run on the IBM PC, including word processing, database, and spreadsheet software. For example, Lotus Corporation developed a version of the spreadsheet Visicalc (originally designed to run on the Apple II) to run on the IBM PC platform. Within a year of its introduction, Lotus 1-2-3 eclipsed Visicalc and became the spreadsheet market leader. Its success led to the label "killer app" to designate an application program of such widespread popularity that it spurs sales of a hardware/operating system platform. This reinforced the importance of owning an IBM PC, thereby adding further to the value of IBM's trademark in the microcomputer market. The powerful IBM trademark and the growing availability of software designed to run on the IBM/Microsoft platform catapulted IBM to a dominant position in the early microcomputer marketplace and greatly encouraged the dissemination of microcomputers. It also made Microsoft and Intel well-recognized trademarks in the microcomputer industry.

Microsoft and Intel retained rights to market their products to other OEMs in the computer industry. Because of the availability of software designed to run on the IBM PC platform, other OEMs sought to develop computer systems that could run the growing supply of IBM-compatible software. Although Microsoft's MS-DOS operating system could be licensed in the marketplace, IBM refused to license its BIOS chip. As a result, other OEMs could not fully emulate the internal operations of the IBM PC readily, and some software designed for the IBM PC did not operate satisfactorily on the computer systems of other OEMs. As a result, consumers strongly favored IBM PCs. Other computer companies had little choice but to offer IBM PC compatibility in order to compete effectively in the microcomputer marketplace. Computer manufacturers that developed their own platform did not fare well.

1. The BIOS chip is the set of essential software routines that test hardware at startup, start the operating system, and support transfer of data among hardware devices. It is typically stored in read-only memory (ROM) so that it can be executed when the computer is turned on. The BIOS is usually invisible to computer users.

With the exception of Apple Computer, which maintained a niche in the marketplace, no serious alternative to the IBM PC/MS DOS platform survived.

By 1984, Compaq developed a BIOS chip that successfully ran software developed for the IBM PC. Later that year, Phoenix Technologies Ltd. developed a fully IBM PC-compatible ROM BIOS, which it licensed to a broad range of OEMs. Other OEMs entered the market for IBM PC-compatible computer systems. As consumers became increasingly confident that software application programs designed for the IBM PC would run on the computer systems of other OEMs, these PC “clone” computers eroded IBM’s dominance of the marketplace by offering lower prices, wider selection, and additional features.

By 1986, numerous OEMs competed in the IBM-compatible/MS-DOS marketplace and IBM’s hold on the market had significantly loosened. The broad range of software available for the IBM-compatible/MS-DOS platform enabled MS-DOS to emerge as the de facto operating system standard in the industry by the late 1980s.

At about the same time, Microsoft began developing the Windows operating system platform incorporating a graphical user interface. The Windows platform was backward-compatible with MS-DOS (i.e., applications designed to operate in the MS-DOS environment could run on the Windows platform as well). Most MS-DOS users as well as new computer users migrated to the Windows platform, which has been the dominant platform since the mid-1990s.

## 2. Computer Software

During the 1940s and 1950s, hardware and software innovation were integrated. The development of computer software was a highly specialized field of scientific research done by academic, government, and government-funded commercial research laboratories. Those who worked with computers had significant scientific and technical expertise. The computer languages and techniques for developing programs were just being created and tested. Computers had relatively narrow use in scientific, military, and space applications. Each computer was unique, and programming was specialized for each machine.

IBM became and remained the dominant force in the commercial computer industry from the 1950s until the early 1980s. During the 1950s and 1960s, IBM and other mainframe manufacturers (e.g., Burroughs, Raytheon, RCA, Honeywell, General Electric, Remington Rand) bundled operating system and application software with hardware for the same price, commonly through a leasing arrangement. During the early stages of the industry, this bundling arrangement made economic sense because there were relatively few computer applications and the hardware manufacturers were able to support these uses of their systems.

As the industry developed, manufacturers encouraged their customers to share software among themselves through software-sharing institutions. IBM

formed and supported a user group named SHARE, which served as a clearinghouse for programming information and software for computer users. SHARE distributed software programs, including libraries of subroutines, algorithms published in technical journals, computer code published in textbooks and, in some instances, programs written to solve problems in specific areas. Those companies contributing to the software-sharing “bank” were entitled to borrow the works of others. But as computers became increasingly powerful, versatile, and affordable, the sharing model began to break down. Those companies making substantial investments in software development were less willing to share these innovations with others. In addition, computer technology was diffusing from government and scientific uses to commercial applications.

Specialty software supply houses, such as Applied Data Research, Inc. (incorporated in 1959), emerged to provide customized and general-purpose software in direct competition with the hardware manufacturers. Offering specialized services on a contract basis, this early software industry competed with the bundled (and, hence, unpriced) software programs provided by mainframe manufacturers through mainframe sales and leases.

The advent of less expensive minicomputers as well as the growing versatility and computing power of mainframes spurred the independent software industry. By 1965, there were approximately 40 to 50 independent software suppliers. F. Fisher, J. McKie, & R. Mancke, *IBM and the U.S. Data Processing Industry: An Economic History* 322 (1983). Applied Data Research introduced Autoflow, a flowchart program, which was the first internationally marketed computer program. International Computer Programs, Inc. (ICP) published catalogs of software programs. The independent software industry grew quickly. There were almost 3,000 vendors by 1968. In 1969, contract programming produced revenues of \$600 million; software products generated another \$20-25 million. *Id.* at 323. Nonetheless, this accounted for less than 10 percent of the amount spent for programming; the remainder was spent on programmers working in-house.

Founded in 1959, Computer Sciences Corporation (CSC) became a successful software company during the mainframe era. CSC began its business by designing, developing, and implementing software systems for computer manufacturers. Over the course of the 1960s, its computer programming business branched out to serve large companies outside of the computer industry and federal, state, and local government agencies. During this same period, CSC increasingly shifted its focus toward the development of software products. It developed a range of products generally directed to business uses, including tax, accounting, and personnel management software.

IBM’s increasing dominance of the computer industry led to antitrust scrutiny by the federal government. In addition, the costs of software development within IBM increased dramatically, and there was increasing pressure within the company to price software separately. Following the lodging of the government’s antitrust complaint in 1969, IBM voluntarily unbundled its



hardware from application programs effective in January 1970. This event greatly expanded the business opportunities for independent software vendors. By 1975, there were over 1,000 software firms in the United States offering more than 3,000 products.

The introduction of the microcomputer in the mid- to late 1970s dramatically changed the software industry. The proliferation of minicomputers fostered the growth of ISVs and a shift away from custom programming and support services toward pre-packaged software products. With relatively small investments, computer programmers could develop software for the growing numbers of microcomputer users. Beginning in the late 1970s, ISVs began selling through retail and other channels pre-packaged (i.e., non-customized) software products for use on microcomputers. Wordstar, Visicalc, and other independently developed software products dominated the early microcomputer software marketplace.

As noted in the discussion of computer hardware, Lotus Corporation developed a version of the spreadsheet Visicalc to run on the IBM PC platform. The powerful IBM trademark and the growing availability of software designed to run on the IBM/MS-DOS platform catapulted IBM to a dominant position in the early microcomputer marketplace and greatly encouraged the dissemination of microcomputers. These factors stimulated rapid growth in the software industry.

By the late 1980s, Microsoft had emerged as a dominant force in the computer industry. Its MS-DOS operating system was installed on the majority of microcomputers and its Windows graphical user interface platform was gaining acceptance in the higher end of the microcomputer marketplace. By 1991, Microsoft's operating systems were installed on almost 90 percent of microcomputers in the world. Building upon this success, Microsoft began bundling its office software products into an office suite of products: Microsoft Word word processing software, Microsoft Excel spreadsheet software, Microsoft Access database software, and Microsoft PowerPoint presentation software. This marketing strategy has enabled it to become the leading seller in each of these product categories.

## **B. AN INTRODUCTION TO COMPUTER TECHNOLOGY**

Virtually unknown 50 years ago, computers literally surround most Americans in their daily lives today. In their most easily recognized form, mainframe and minicomputers can be found in most businesses, government offices, and schools. Microcomputers can be found on most business and home desktops for use in word processing, information storage, entertainment games, and electronic shopping. Less commonly recognized, computers can also be

found in many home appliances, hand-held organizers, telecommunication devices, automobile dashboards, and elevators, among other places.

## 1. Computer Hardware

Computers use a binary base. By setting electrical switches to “on” (electrical current is flowing) or “off” (current is not flowing), early computers could create a single “bit” of information. That piece of information is read as either a 1 (“on”) or a 0 (“off”). By translating information into a series of such 1s and 0s, computers could perform mathematical operations.

The first computing machines did not utilize computer “programs” in a form that we would recognize today. These machines were in essence a series of hard-wired circuits constructed to perform one particular computational task. That is, the mathematical function performed by the computer was determined by the physical arrangement and structure of the circuits. The computers had to be rewired in order to perform a different function. These machines were comprised solely of what we call today “hardware”—the physical circuits that make up the machine.

During the late 1940s, scientists developed the first machines that could store and use encoded instructions or programs. This set of innovations dramatically increased the flexibility and usefulness of computers. Users could perform a variety of computational tasks without having to rewire the basic hardware of the computer. Instead, they could simply direct the computer to perform one of the functions that it had stored in its memory. The actual computer in these programmable or “universal” machines is the central processing unit (CPU). The CPU has two principal components: an arithmetic logic unit (ALU), which performs a basic set of primitive functions such as addition and multiplication, and a control unit, which directs the flow of electric signals within the computer. In essence, a computer processes data by performing controlled sequences of primitive functions. As computers grew more powerful and the tasks they performed grew more complex, computer scientists relied on increasingly complex sets of instructions that are executed automatically by the computer. These sets of instructions, known as computer programs, are the focus of most of this book.

The basic hardware of a modern microcomputer system includes a CPU, internal memory storage, disk drives or other devices for physically transferring data and programs into and out of the internal memory, and telephone or network interconnections for linking the computer with other computers. The internal memory of the computer typically features three types of information storage: random access memory (RAM), read-only memory (ROM), and data storage memory (disk space). Data can be input into RAM, erased, or altered. RAM chips serve two information storage functions: they act as temporary storage devices for programs and data currently running on the computer, and they also serve as permanent memory for data or programs. A ROM chip has information permanently embedded in the architecture of the chip, and that

information can only be read (not altered) by the computer. ROM chips are used primarily to direct certain basic operating functions of the computer.

Computer engineers design the programming capability of a computer to suit the user's needs. By building more of the desired functions directly into the wiring of the computer, they can achieve more efficient processing for certain applications. Such pre-designed computers are known as "special purpose computers" because they are designed to perform only certain specific tasks. Their greater speed comes at a cost of less flexibility—that is, less ability to run a wide range of programs. This technological trade-off harks back to the early days of computer technology, when all programs were hard-wired into the computer.

Advances in computer technology have made greater efficiencies of processing possible without the need to hard-wire the computer. Most modern computers, particularly personal computers, are "general-purpose computers" that feature a high degree of programming flexibility. When a user has only a few computing needs or desires high-speed processing, however, she may still prefer to rely heavily on internal programming.

Besides the internal memory and processing chips, computers are composed of input and output devices (sometimes referred to as I/O devices) and peripherals. These devices control the transfer of information into and out of a computer. Early programmers "input" information into a computer by changing the physical structure of its circuits, or (in more sophisticated models) by using "punch cards," which allowed computer users to write data for the computer in the form of holes punched in special note cards and then to feed that data into the computer in the form of stacks of such cards. Most modern computers use the typewriter keyboard as their primary input device, allowing users to enter data into the computer by typing it. The typewriter keyboard has been supplemented with other input devices, including the computer "mouse," the telephone line, and microphones coupled with voice recognition software.

The output devices of computers have also changed. Computers originally communicated data to humans in the form of lights that turned on or off, representing the bits of data produced by some computer operation. Advances in computer outputs include the development of the printer, the introduction of television-like computer monitors and screen displays, and telephone and cable output that can "send" data to a remote location.

## **2. Computer Software**

Computer programs are the instructions that allow general-purpose computers to be many different types of machines. When a computer is running a video game, the computer *is* a video game machine. When it executes program instructions to enable users to write letters or reports, the computer *is* a word processing machine. When it carries out a programmed search for data in a large repository of information, the computer *is* an information retrieval machine. Programs can also be written for special-purpose hardware (e.g., the semi-

conductor chip that monitors the functioning of your toaster) when this will best achieve the developer's objectives.

Computer programs that operate on a single machine fall into two basic categories: operating systems programs and applications programs. Operating system programs manage the internal functions of the computer. They coordinate the reading and writing of data between the internal memory, the CPU, and the external devices (e.g., disk drives, keyboard, and printer); perform basic housekeeping functions of the computer system; and facilitate use of application programs. In essence, the operating system prepares the computer to execute the application programs and serves as an intermediary between the application program and the hardware of the computer. An applications program can order the deletion of a file, but, generally, the operating system actually carries through the details of this function.

Every computer needs an operating system to direct its functions and to manage other software that is run on the computer. Computers do not need, and many do not have, more than one operating system. Because the operating system controls the interactions between the user, the software, and the computer itself, an applications program must be compatible with a particular operating system if it is to interoperate with that program and run on a computer using that operating system. This compatibility requirement means that the designers of operating systems have some degree of control over the applications programs that will work with that operating system. Although the specifications of applications programming interfaces (APIs) are sometimes published freely, often they are licensed from an operating system program developer. Microsoft, for example, licenses its APIs for its operating system programs to applications developers. An alternative way to get access to APIs is through a laborious process of reverse engineering the program (about which more in Chapter 2).

Operating systems may control the execution of instructions in the central processing unit of the computer, but they generally do not perform specific tasks of interest to end users. Applications programs enable users to accomplish specific tasks with computers. Bookkeeping, statistical and financial analysis, word processing, and video game programs are among the many types of application programs available today.

Application programs are often developed to run on particular operating systems. The task of adapting an application program designed to run on one operating system to run on another operating system is often technically complex, time-consuming, and costly. In recent years, software developers have developed programs that run on more than one operating system. In addition, translator programs have been developed to allow users to move files from one application program to another. Nonetheless, the problem of compatibility between applications programs and operating systems remains a major concern in the computer software industry. There is some hope that the Java programming language (about which more in Section C) will enable programmers to write a program once and have it run on many machines.

The line between operating systems and application programs, while sharp in particular instances, may blur when programs once distributed as applications programs are integrated into an operating systems program. Microsoft, for example, has integrated a number of “add-on” features (such as compression software and even rudimentary word processing) into its operating system over time. By selling a bundled product that includes both an operating system and certain applications, Microsoft arguably provides more value to consumers but also eliminates a competitive market for such add-on products. Microsoft’s decision to make its Internet Explorer Web browsing software an integrated part of the Windows operating system contributed to the U.S. Justice Department’s decision to charge Microsoft with antitrust violations in the late 1990s. (We discuss this suit in more detail in Chapter 7.)

### **3. Computer Networks**

Early computers were self-contained machines. Data could be input into them (usually laboriously by punch card), and after the computer processed the data, output would be produced. But the data never left the physical environs of the single computer. To move data from one computer to another, one had to take the output of one computer and transport it physically to and then input it into the new computer. Even when input and output became somewhat more efficient—for example, by use of magnetic storage media such as “floppy disks”—the necessity for a physical transfer remained.

Computer engineers recognized early on that great benefits would flow from the networking of computers. Networking has been around in specialized computing environments since the late 1960s, but it was not until the 1990s that the Internet afforded easy widespread access to large computer networks. As of June 1999, more than 171 million people around the world had Internet access and 37 percent of the U.S. population had Internet access at home or at work.

Networking technology allows computers to communicate with each other. This communication capability enables dispersed computer users to exchange information—for example, by electronic mail (email). It also reduces the time and cost of transferring information regardless of the physical location of computers. Networked computing also allows computers—and people—to work together to achieve certain tasks that would take much longer to do alone.

Networking computers requires some basic technologies. First, some form of hardware must connect the two computers, either physically or virtually. If the computers are physically close to one another, this can be accomplished by stringing a cable between them. If the computers are physically separated by great distances, networking requires that they be connected either via telephone lines or by some form of wireless communication. Long-distance connections require some form of hardware to convert digital computer data into a form that can be transferred over the telephone lines or “narrowcast” over the airwaves. One common type of hardware that enables telephone transmissions of digital

data is a modem. Modems take digital computer data, convert it to analog (sound) form for transfer over telephone lines, and then reconvert it to digital form. Networking software (and sometimes hardware components as well) is required to enable computers to communicate with each other. When one computer sends data, the other must be able to process it. Interoperability issues thus arise in the context of computer networking design. The development of standard protocols for exchanging information has fostered the growth of networking.

**Local Area Networks.** Many organizations find it useful to develop local area networks (LANs) to link a number of computers so that members of the organization can share information and information resources. LANs provide all of the communications advantages described above: they enable employees to communicate by email, to send files to one another, and to share computing resources. In addition, LANs have made possible a form of specialization or division of labor among computers. Because the computers in a LAN are linked together in real time, it is possible to store files in a single central location and allow any computer to access them at any time. Rather than requiring each computer to be self-sufficient, certain (generally more powerful) computers can be designated as central “servers” where files or programs are located. Individual “client” computers in the network can call up the files or programs on an as-needed basis. Because the server machines are generally faster and more powerful than ordinary computers, LANs offer not only centralized access but also quicker processing time.

LANs also facilitate group projects. “Workgroups” can add to or change the same document simultaneously over the network. This is a particular advantage for companies (e.g., sales companies, airlines, hotels) that rely on large information databases that must be regularly updated. Documents or databases that previously had to be changed by one central programmer can now be altered instantaneously to reflect current information. Because the workgroup is organized on the computer network rather than by the physical sharing of documents, LAN workgroups also contribute to a more flexible organizational structure.

The mass corporate movement towards LANs has important legal implications as well. Information that used to reside on a single computer is now accessible to dozens or hundreds of computers, many of them physically remote from the actual location of the data. LAN administrators must worry about limiting access to their networks, guaranteeing the security of their data, implementing version control on revised documents, and monitoring access to certain “controlled” data (particularly applications software licensed for limited uses from third parties).

Other kinds of private networks also exist. Anyone with a computer and a modem can connect to private dial-up networks. Such dial-up connections normally take place over the telephone lines between individual computers and one or more central “host” computers operated by the private network

administrator. Some are operated by information providers who allow people to sign on to the network and download information of particular interest to them. Others may serve as communications forums for people interested in particular topics. Individuals dial into the host computer and communicate with each other through the host, either in real time or by leaving email messages.

**Large-Scale Public Networks.** In the mid-1960s, researchers working for the Department of Defense's Advanced Research Project Agency (ARPA) began working on the problem of connecting its computers scattered around the United States at various universities and research laboratories to enhance memory storage and time-sharing capabilities. An important design objective of this system was that it not be vulnerable to breaking down in the event of a nuclear attack or other widespread disruption of telecommunications systems. It was initially believed that this would require 136 separate communication lines ( $17 \text{ computers} \times 16 \div 2$ ). These lines would be expensive, and the cost would grow geometrically as more computers were added to the network.

At about that time, researchers at Rand Corporation and the National Physical Laboratory in England independently developed the concept of "store-and-forward packet switching" which avoided the problem of having to connect independently each node of a network to each other node. Packet switching technology, using techniques similar to those developed in the telegraph industry, allows multiple messages to flow through a common "backbone" line by transmitting information in smaller packets that contain the address of the destination. Large messages are broken into streams of packets that are sent as individual items into the network. Switching nodes pass the messages along and the receiving computer reconstitutes the full message. Such a system minimizes the risk that the entire system could crash by allowing information to travel along a variety of paths.

In 1970, the ARPANET successfully implemented packet switching technology in a four-node network. The ARPANET grew to more than 20 sites by 1971 and to over 200 sites by 1981. The ARPANET paved the way for the Internet, an international collection of *interconnected* computer *networks* based on an open technical standard known as Transmission Control Protocol/Internet Protocol (TCP/IP). Computers implementing this protocol may connect to the Internet.

By 1996, more than 9 million host computers were connected to the Internet, and it is projected that this number will exceed 200 million by the year 2000. Computer users may gain access to the Internet through Internet service providers (ISPs) such as America Online (AOL), Microsoft Network (MSN), and Netcom; university and library connections; corporate and non-profit portals; and government nodes. Thus, the Internet is a largely decentralized system utilizing a common communication backbone.

Until recently, the Internet has been governed primarily by InterNIC (NSFnet Internet Network Information Center), a consortium involving

the National Science Foundation, AT&T, General Atomics, and Network Solutions, Inc. (NSI). NSI has had principal authority to register Internet names and addresses, what have come to be known as “domain names.” Domain names are represented in a hierarchical format: server.organization.type—e.g., www.whitehouse.gov. In 1999, the assigning of domain names was opened up to a range of entities operating under the authority of the Internet Corporation for Assigned Names and Numbers (ICANN).

In 1990, Tim Berners-Lee, a researcher at CERN, a European particle physics laboratory, developed the World Wide Web (WWW or Web), a widespread means by which users of the Internet could share databases. The WWW makes accessible to Internet users hypertext documents residing on HTTP (hypertext transfer protocol) servers throughout the world. These Web pages, identified by uniform resource locators (URLs), such as www.whitehouse.gov, are written in HTML (hypertext markup language). Codes embedded in Web pages can instantly access other documents on the World Wide Web. They may also activate embedded software programs and audiovisual images. The Web became a mass media phenomenon with the development of “Web browser” programs that permit personal computers to access a wide variety of files from Web sites. The Web has also become the transaction medium for most electronic commerce.

Users of the Internet can locate Web sites of interest in a number of ways. They can often find on-line merchants, educational and government entities, and companies by using trade and organization names, such as “Microsoft.com,” “law.berkeley.edu,” and “uspto.gov.” They can also “surf” the Internet with the assistance of various “search engines,” such as Yahoo, Altavista, and Infoseek. These search engines look for keywords in domain names, actual text on Web pages, and metatags—HTML code created by a Web page developed to attract search engines looking for particular keywords. Search engines will then rank Web sites from all over the Web according to various algorithms designed to arrange indexed materials. Web users may create “bookmarks” on their computers in order to access their favorite Web sites quickly.

As computing has shifted in the 1990s from a stand-alone activity to one conducted increasingly over networks, the way computers function has changed as well. It no longer makes sense to talk about how *a* computer functions. Most computers used in business, and many used in the home, do not operate in self-contained fashion. Computer programs are distributed across LANs and over the Internet, and over the Internet more and more programs are distributed in pieces on an as-needed basis. We discuss this change and its implications for computer law in more detail in the sections that follow.

The Internet has also changed the way commerce is conducted. Virtually everyone was caught off guard by the speed with which the millions of people using the Internet began to use it to buy things. The development of such “ecommerce” poses important new challenges for the legal and technological framework of the Internet. We discuss those challenges in detail in Chapter 9.



## C. HOW SOFTWARE IS MADE

Software development has evolved from a significantly hardware-constrained activity to a highly flexible and sophisticated field of engineering. This section briefly summarizes the principal methods involved in modern software development. We note, however, that the proliferation of a wide range of computers—from mainframes to desktop units to a host of embedded systems—has spawned a great range of programming methods. Although many of the stages can be found in other programming contexts, we focus here upon the methods used in the development of relatively complex commercial operating systems and application programs.

Software development is an inherently functional enterprise. Software provides the instructions that enable a computer to perform tasks that serve the users' needs. Software can control anything from the relatively simple operation of a clock to the highly complex operation of an airplane. Whereas the programs that operate a wristwatch may have 1,000 instructions, modern spreadsheet and word processing programs have well over a million lines of instructions or code. In order to make computers more effective and easy to use, software can become extraordinarily complex in design and implementation. To a large extent, software engineering has become an applied science of mastering complexity. The engineering nature of the discipline is discussed in detail in Pamela Samuelson, Randall Davis, Mitchell D. Kapor, & J.H. Reichman, *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 *Colum. L. Rev.* 2308, 2327-2329 (1994) [hereinafter *Manifesto*].

### 1.4.3 Constructing Programs Is an Industrial Design Process

Once one understands that programs are machines that happen to have been constructed in the medium of text, it becomes easier to understand that writing programs is an industrial design process akin to the design of physical machines. Each stage of the development process requires industrial design work: from identifying the constraints under which the program will operate, to listing the tasks to be performed (i.e., determining what behavior it should have), to deciding what component parts to utilize to bring about this behavior (which in the case of software includes algorithms and data structures), to integrating the component utilitarian elements in an efficient way.

A substantial amount of skilled effort of program development goes into the design and implementation of behavior. Designing behavior involves a skilled effort to decompose the overall, complex task (e.g., word processing) into a set of simpler subtasks requiring simpler behaviors (e.g., deleting a character). Constructing the interaction of these subtasks to produce useful behaviors (e.g., deciding whether the “delete word” command should be implemented as a sequence of delete character commands) also requires design skill. Knowing how and where to break up a complicated task and how to get the simpler components to work together is, in itself, an important form of the engineering design skill of programmers.

The goal of a programmer designing software is to achieve functional results in an efficient way. While there may be elements of individual style present in program design, even those style elements concern issues of industrial design, e.g., the choice of one or another programming technique or the clarity (or obscurity) of the functional purpose of a portion of the program. . . .

At the same time, others have characterized programming, particularly in its earlier days, as akin to an art form. Certainly it is true that while the ultimate goal of computer programming is the design of a functional work, programming has historically been less routinized than most engineering disciplines, and much computer programming has involved “reinventing the wheel.”<sup>2</sup> We discuss some of the reasons for that, and modern ways of dealing with it, in the sections that follow.

Software development has traditionally been described as a multistage process often analogized to a waterfall.<sup>3</sup> See generally Stephen R. Schach, *Classical and Object-Oriented Software Engineering with UML and C++* (4th ed. 1999); Ben Schneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (3d ed. 1998); Grady Booch, *Object-Oriented Analysis and Design with Applications* (2d ed. 1994); Ian Sommerville, *Software Engineering* (4th ed. 1992). All software development processes begin with a clear definition of the problem to be solved or task to be automated. The process begins with the identification of the goals of the users and the constraints of the hardware system. This stage is followed by the development of a system and software design. A user interface will also be designed to serve the needs of the target audience for the software product. After the software has been structured and an interface designed, programmers implement the design, test its performance and reliability, and fine-tune the system. In addition, many software products require ongoing maintenance and updating to correct errors and enhance their capabilities.

Although the waterfall model implies a linear flow, modern software engineering typically has an iterative quality with many feedback loops along the development and use life cycle. That is, programming does not occur from the top down, with ideas being turned into algorithms and thence into code. Rather, the development of lower-level program components can influence the design of higher-level components and can even cause the programmer to rethink the goals of the program itself. One might think of this as bottom-up programming. As the authors of the Manifesto note:

Innovation in software development is typically incremental. Programmers commonly adopt software design elements—ideas about how to do particular

2. The Manifesto observes that “[t]ypically, a programmer writes every line of code afresh, no matter how large the program is, or how common its tasks. To perceive the impact of this lack of standard building blocks, imagine trying to design an entire car in complete detail, down to the last fastener, without being able to assume the existence of any standard parts at all (not even nuts, bolts or screws).” As its authors observe, “[t]his is not a desirable state of affairs.” Manifesto, *supra* at 2322.

3. Other programming paradigms include the rapid prototyping model, exploratory programming, formal transformation, and system assembly from reusable components.

things in software—by looking around for examples or remembering what worked in other programs. These elements are sometimes adopted wholesale, but often they are adapted to a new context or set of tasks. In this way, programmers both contribute to and benefit from a cumulative innovation process. While innovation in program design occasionally rises to the level of invention, most often it does not. Rather, it is the product of the skilled use of know-how to solve industrial design tasks. . . .

The technical community has recognized the cumulative and incremental nature of software development, and has welcomed efforts to direct the process of software development away from the custom-crafting that typified its early stages and toward a more methodical, engineering approach. The creation of a software engineering discipline reflects an awareness that program development requires skilled effort and applied know-how comparable to other engineering disciplines.

The products of software engineering almost invariably contain admixtures of old and new elements. The innovation in such programs may lie in the manner in which the known elements have been combined in a new and efficient manner. Or it may come from combining some new elements with well-known elements in order to achieve the same result in a new way. When we speak of programs as “industrial compilations of applied know-how,” it is in recognition of the frequency with which software engineering involves the reuse of known elements. Use of skilled efforts to construct programs brings about cumulative, incremental innovation characteristic of engineering disciplines. A well-designed program is thus akin to the work of a talented engineer whose skilled efforts in applying know-how, accumulated from years of experience and training, yields a successful design for a bridge or other useful product.

Manifesto at 2330-32.

However programs are designed and written, a fundamental fact about computer programs is that they are functional. They are designed to carry out certain tasks and bring about certain behaviors. Consequently, programs are judged largely on how well they perform the tasks they have been programmed to accomplish. Because the computer instructions serve no function other than to accomplish the defined tasks, the overriding concern in designing a program is to meet the users’ needs in the most efficient manner.

The concept of efficiency in this context is broad. “Efficiency” may mean one or more of the following: (1) code efficiency—maximizing the processing speed; (2) memory efficiency—using solution techniques and addressing methods to minimize the amount of memory needed to accomplish the desired tasks; (3) input/output efficiency—maximizing the quality and speed of information transmission between the computer and the user or external hardware devices (such as keyboards and printers); (4) stability—the program must be easy to maintain, upgrade, and adapt to new hardware platforms; and (5) usability—the ease of use by the intended audience. The software engineering field strives to develop methods for improving the efficiency and reliability of programs in a cost-effective manner.

The subsections that follow describe typical processes for designing software at different levels of abstraction, from high-level ideas down through the actual writing of program code.

## 1. Requirements Analysis and Program Specification

At the conceptual level, software development is comprised of several tasks. Of most importance, the design team must identify the goals of the design process. For example, in developing software to provide banking services through an automated teller machine (ATM), the design team will map out the necessary data, the desired functionality, and hardware and security constraints. They might develop tables or flowcharts to understand the flow of information needed to accomplish the various transactions. As another example, in developing software to run a law office, the designers must identify the various tasks that the computer program should handle—for example, billing clients, filing documents, ordering supplies, preparing budgets, and filing court documents. The design team would typically interview the prospective users of the software system about their needs and desires, study the way information flows in a law office, and develop a schematic representation of the tasks to be programmed. This abstract representation of tasks maps data inputs and outputs and assesses the hardware requirements for possible systems.

Writing an application program is a complex and iterative process. While some programs are small and relatively simple to write, most modern programs involve thousands and even millions of lines of computer code. They may be written not by a single person but by teams of programmers working together over the course of months or years.

Partly to facilitate collaborative work within teams, programmers sometimes develop a flowchart to depict the logical structure of the program. This will not just show what the program is supposed to do but also how it will carry out the desired tasks. A sample flowchart for a computer program is reproduced as Figure 1.

We should emphasize, however, that flow charts are merely conceptual aids, and are by no means a necessary part of programming.

## 2. Software Design

Processes for designing software systems have undergone significant changes over the past four decades as computer hardware has become more powerful and versatile and the problems sought to be addressed have become more complex. Managing the complexity of large-scale software projects—for example, computer operating systems, avionics (programs to fly large commercial airplanes), robotics, spacecraft simulation, telecommunications, air traffic control—entails significant technological creativity. Programmers have traditionally sought to divide complex problems into component parts and to solve each component separately. This procedure-oriented, “functional decomposition,” or “top down” methodology served as the dominant paradigm for software design through at least the mid-1980s, and it continues to be widely used in solving some classes of problems. The programmer begins with a general description of the functions that a program is to perform. The

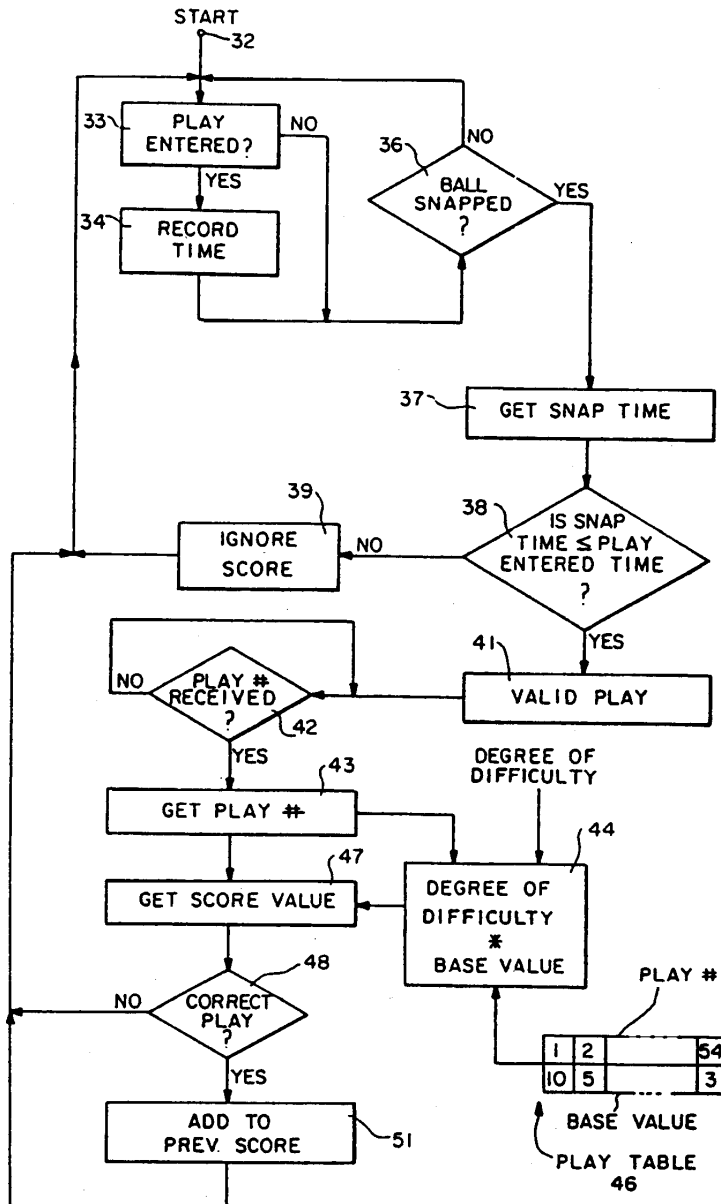


FIGURE 1  
Flowchart from a software patent drawing.

programmer then outlines the program, specifying data structures and algorithms to be used. Such outlines are frequently expressed as flowcharts showing the relationship among the various modules or subroutines of the program. These modules are then separately further broken down until the full logic of the program can be spelled out.

While conceptually logical, the classical design methodology has become increasingly problematic as the complexity, scale, and need for updating of software projects has grown. The many parts of very large software systems often interact in a multitude of intricate and subtle ways. Classical top-down designs often require significant redundancies and are difficult to evolve. Programmers must typically build programs from scratch, often requiring re-invention of many components. This contrasts with other engineering disciplines, which typically reuse existing components. Such reuse can reduce design, testing, and other costs. In addition, top-down procedural techniques typically define data structures in one place. Various subroutines refer back to this single location. While this approach offers some efficiencies, it can lead to problems when the program is updated or revised. Whenever a data structure is revised, all subroutines or modules drawing upon that data structure must be suitably altered as well, which can cause a domino effect.

As a result of these limitations of classical programming methodologies, which tend to become more acute as systems grow large, computer scientists developed the “object-oriented” paradigm. Programmers using this paradigm design software by structuring relationships among independent “objects,” each of which represents a physical entity in the real world. Each object stores both data representing attributes of the physical objects as well as procedures representing actions that the physical object can perform. Whereas the traditional top-down program is structured around processing of data, object-oriented programs model a problem by actually simulating the interaction of physical entities. Objects are grouped within hierarchical structures. Higher-level classes “inherit” the attributes of subclasses. Object-oriented systems simplify complexity by “encapsulating” the internal data structures and procedures within objects. The data structures and procedures of particular objects can be altered without having to affect other aspects of the larger software system.

Object-oriented programming reflects basic human learning processes. Grady Booch, one of the pioneers of the object-oriented paradigm, offers the following analogy:

[W]ith just a few minutes of orientation, an experienced pilot can step into a multi-engine jet aircraft he or she has never flown before, and safely fly the vehicle. Having recognized the properties common to all such aircraft, such as the functioning of the rudder, ailerons, and throttle, the pilot primarily needs to learn what properties are unique to that particular aircraft. If the pilot already knows how to fly a given aircraft, it is far easier to know how to fly a similar one.<sup>4</sup>

4. Grady Booch, *Object-Oriented Design with Applications* 12 (1991).

Thus, a programmer starting with an object-oriented program that simulates a basic aircraft can simulate the behavior of a more sophisticated aircraft, such as a fighter jet, by adding additional features, behaviors, and processes of the more advanced plane. The programmer would not need to begin the software design process from scratch.<sup>5</sup>

The above example illustrates some of the ways in which object-oriented design systems economize on programming time and cost. Such designs are more readily adaptable to changes in data structures and new variables than are the traditional top-down approaches. The adaptability of object-oriented software programs substantially reduces the risk that a large investment in software design will be lost if new parameters or features need to be added to a program. Moreover, object-oriented designs tend to yield smaller systems through the reuse of common mechanisms, which reduces the complexity of the software design and the cost of writing code. See Mark A. Lemley & David W. O'Brien, *Encouraging Software Reuse*, 49 *Stan. L. Rev.* 255, 259-268 (1997). This reuse of "chunks" of computer code is not only valuable because it saves the cost of rewriting the code, but because it permits a particular software "object" to be refined and debugged and then reused in a variety of different programs. This reuse of software objects by different firms may enhance software compatibility, decrease programming costs, and even improve the safety and reliability of software (since mission-critical software will not have to be written from scratch for each new product, but can use code that has already been tested and proven to work). As a result of these advantages, object-oriented methodologies have increasingly become the norm in designing highly complex computer programs.

### 3. User Interface Design

As computers have become more versatile and available to a broader and often less technically trained range of users, software development has increasingly focused upon tailoring the program to the particular goals and knowledge base of the intended users. See generally Ben Schneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (3d ed. 1998). The design of computer program user interfaces draws upon the field of computer-human interaction (CHI), a sub-field of a more general field of "human factors" analysis that aims to understand how human beings process information so that products can be better designed to enhance usability. Human factors analysis brings together insights from the fields of education, graphic art, industrial design, industrial management, computer science, mechanical engineering, psychology, artificial intelligence,

5. See Barkan, *Software Litigation in the Year 2000: The Effect of Object-Oriented Design Methodologies on Traditional Software Jurisprudence*, 7 *High Tech. L.J.* 315 (1992); Smith, *Abstraction, Filtration, and Comparison in Computer Copyright Infringement: An Explanation and Update for the Object-Oriented Paradigm*, 26 *AIPLA Q.J.* 1 (1998).

linguistics, information science, and sociology. Recognizing the many ways that the study of human factors can aid computer system design, the computer industry has taken a particularly strong interest in the effort to synthesize and expand this learning.

The field of computer-human interaction has profoundly changed the way in which application programs are both conceptualized and written. The field has identified five human factor goals that programmers should strive to achieve in designing application programs: (1) minimize learning time, (2) maximize speed of performance, (3) minimize rate of user errors, (4) maximize user satisfaction, and (5) maximize users' retention of knowledge over time.

Application programmers attempt to achieve these objectives in designing computer-human interfaces. As a result, many of their design choices are guided by principles of human factor analysis that have evolved over years of empirical research. Among the design issues that have been studied are: the choice among command-based programs, menu-driven programs, and natural language programs; menu design; windowing versus scrolling; the choice of abbreviations and command names; the layout and scheme of graphical interfaces; the color and highlighting of video displays; the consistency of user interfaces; and the design of direct manipulation interfaces. To aid in the implementation of these ideas, many firms engaged in application program development have compiled computer-interface guidelines and criteria to guide their programmers. Apple Computer, which introduced the first commercially successful graphical user interface, developed the following guidelines for user-friendly software design:

1. *Metaphors*—the interface should embody plain, simple metaphors for accomplishing tasks with audio and visual effects to support the metaphor.
2. *Direct Manipulation*—the interface should allow users to directly manipulate items or actions on the screen, rather than indirectly through abstract commands.
3. *See and Point*—the interface should allow the user to effect action by a see-and-point mechanism that is intuitive, rather than by a remember-and-type mechanism that can be intimidating, too abstract or unnatural.
4. *Consistency*—the symbols and mechanisms for accomplishing tasks should be consistent across all application programs through a consistent interface.
5. *WYSIWYG (What You See Is What You Get)*—the interface should present the information on the screen to the user in exactly the form that it will come out on the printer, removing the need to type in abstract formatting commands or to make mental calculations to envision how the screen translates to paper.<sup>6</sup>

These principles—and the basic concept of graphical user interfaces—are in widespread use in the software industry and can be seen in most application programming and web-based user environments.

6. Written submission of Apple Computer, Inc., U.S. Copyright Office Public Hearing on Registration and Deposit of Computer Screen Displays 5-6 (Sept. 4, 1987).



## 4. Generating Computer Code

After designing computer systems, including the user interface, a software engineer engages in a series of steps to translate the design into binary code (representing open and closed circuits) that is actually “readable” by computers. While it is possible to write programs directly into machine-level language, most programming is done in higher-level languages that use abbreviations and short words to convey the action the program will need to carry out. Well-known programming languages include FORTRAN, COBOL, and PASCAL. Special languages, such as C++, Ada, Smalltalk, Object Pascal, and Java, have been developed more recently to implement object-oriented designs. These programming languages, which can be readily understood by skilled programmers, are often referred to as “source code.”

Figure 2 contains the source code version of a simple program written in PASCAL for determining and listing in five columns any number of prime numbers. PASCAL is a high-level programming language, and, as you can see, it is relatively easy to comprehend. For example, the statement “Go to 40” instructs the computer to skip intervening steps and execute the instruction at line 40. But it is a specialized language all the same, and (because the programmer is writing for a computer audience) its rules must be followed precisely. “Execute line 40 now” may mean the same thing to a human reader as “Go to 40,” but it may be incomprehensible to a computer.

Traditionally, source code was written by computer programmers who had learned one of the specialized computer languages described above. Increasingly, however, the process of writing source code has itself become automated. Computer programmers can now “write” code with the help of a computer “wizard” that translates higher-level design concepts directly into code. Having a computer help write source code can be faster and more efficient than writing it by hand; it can also help prevent programmers from making mistakes that will interfere with the program. As more and more code is written with computer assistance, similarities in actual program code become less meaningful as evidence of copying. This will become important when we consider copyright law in Chapter 2.

In order to be executable by the central processing unit of a computer, source code must be transformed into a machine-executable form. This is generally accomplished by using a special purpose computer program known as a “compiler” to process source code instructions into “object code,” the binary code that is processed by the computer. In this machine language, a single 0 or 1 represents the absence or presence of an electrical charge. Each binary digit is referred to as a “bit.” These electrical signals processed by a CPU will determine which functions of the CPU will be executed in what order. Computers do not process program instructions one bit at a time. Rather, they process standard lengths of bits. A “word” is the set of characters that occupies one storage location and is treated by the computer circuits as a unit and transported as such. Until the mid-1980s, most computers acted upon words of eight bits,

```

      Program PrimeNum (Input, Output);
    {
    .....
    WRITTEN BY:   Vance F. Brown
    DATE:        April 5, 1988
    PROGRAM PURPOSE: This program allows user to enter the desired number of prime numbers
    to be ascertained. The program then lists the prime numbers (starting with the prime number 2) in
    numerical order in five columns. DEFINITION OF PRIME NUMBER: A number is prime if it
    is not evenly divisible by any number other than itself and 1.
    .....
    }
    Var {for variable function explanation, see below}
        num, count, primes, times, divisor: Integer; check: Boolean;
    Begin {of program}
        {***** initialize variables *****}
        num := 2; {the primenumber candidate}
        count := 0; {number of prime numbers found}
        times := 0; {number of columns before a carriage return}
        {***** ask user how many prime numbers to list *****}
        write ('How many prime numbers do you want to find?');
        read (primes);
        writeln;
        writeln; {carriage returns}
        writeln;
    repeat {until the number of primes is found}
        check := false; {initialize: assume first that number is not
                        prime}
        divisor := 2; {first number to be divided into candidate}
        divisor := succ(divisor); {first number to be divided into candidate}
        {***** loop to determine if numbr is prime *****}
        while (divisor < num) and (not check) do begin

            check := num mod divisor = 0; {the remainder after dividing
                                          divisor into num}

            divisor := succ(divisor); {increment divisor}
        end; {of while loop}

        {***** if a prime number, then display it *****}
        If not check then begin
            write (num:10);
            count := succ(count); {increment number of primes
                                  found}
            times := succ(times); {increment column number}

        {***** column adjustment, once five numbers displayed,
        carriage return*****}
            if times = 5 then begin
                writeln; {carriage return}
                times := 0; {start counter for column number over}
            end; {of column adjustment if statement}
        end; {of display of if prime number statement}
        num := succ(num); {next candidate for prime number}
    until count = primes; {once the number of primes entered by user is
                          found, the program ends}
    end. {of program}

```

FIGURE 2  
Source code in PASCAL, by Vance Brown.



translates assembly language into object code. Figure 4 contains 29 of 7,330 lines of assembly code produced by “disassembling” the object code. Like object code and unlike higher-level languages, assembly language is specific to a particular computer system.

## 5. Software Validation and Maintenance

Although this discussion may suggest that programming is a linear process, it is in fact typically an iterative process. Program design often occurs in a series of “feedback loops” in which each stage of the process influences all of the others. Thus, a software engineer may develop a design for a program and then write some code to implement it, but in the process of writing the code, the engineer may discover logical problems with the design that require rethinking the program’s basic structure. Often, individual modules or components of a software program will be tested and refined before integrating the various components into a full working version.

Before a software program enters service or the marketplace, the software design team runs the program through a suite of tests selected to ensure that the program operates properly and achieves the objectives set out in the requirement and specification stages of the process. After the various systems within the program are operating properly, the team conducts alpha (acceptance) testing. “Alpha testing” involves running the software with actual data (rather than simulated data) from the intended user. For products that are to be marketed widely, software vendors typically run a further level of testing (commonly referred to “beta testing”) in which the program is provided on a limited basis to a range of target customers who agree to use the program and report problems. Beta testing may also identify features that users desire that were not initially included in the program. Based upon this input and further internal testing, the vendor typically makes some further refinements before releasing the product into the commercial marketplace.

All complex software programs have a range of errors (commonly referred to as “bugs”), and the validation stage can be one of the more challenging phases of the software life cycle. Because of the importance of reaching the market quickly, the design team is often under tremendous pressure to complete this phase.

Following the release of a product, software enters the last and what is often the longest phase of its life cycle: maintenance. Computer software users place great significance on a software vendor’s ability to correct, improve, and adapt software products and services. Indeed, consumers are often wary of buying the first version of a new software product because it may be “buggy.” The planning for maintenance begins early in the process with the documentation of the computer program. Most companies put out a series of “updates” or new releases of their programs to address maintenance problems and to enhance the program’s functionality.

Some computer bugs may not manifest themselves until the product has been in use for a long time. The year 2000 (Y2K) problem is to a large extent a

```

L05CF:  PUSH    AX      ;05CF 50
        PUSH    CX      ;05DO 51
        MOV     CL,4    ;05D1 B1 04
        SHR    AX,CL    ;05D3 D3 E8
        ADD    BX,AX    ;05D5 03 D8
        POP    CX      ;05D7 59
        POP    AX      ;05D8 58
        AND    AX,0F    ;05D9 25 0F 00
        RET_NEAR      ;05DC C3
        ;L05DD L062A CC
L05DD:  CMP     BX,DX    ;05DD 3B DA
        JNZ    L05E3    ;05DF 75 02
        CMP    AX,CX    ;05E1 3B C1
        ;L05E3 L05DF CJ
L05E3:  RET_NEAR      ;05E3 C3
        ;L05E4 L0656 CC
L05E4:  ADD    AX,CX    ;05E4 03 C1
        ADD    BX,CX    ;05E6 03 DA
        JMP    SHORTL05CF ;05E8 EB E5
        ;L05EA L0625 CC
L05EA:  MOV    AX,ES:[DI+4] ;05EA 26 8B 45 04
        MOV    BX,ES:[DI+6] ;05EE 26 8B 5D 06
        PUSH   AX      ;05F2 50
        OR    AX,BX    ;05F3 0B C3
        POP    AX      ;05F5 58
        RET_NEAR      ;05F6 C3

```

FIGURE 4  
Assembly code, by Vance Brown.

massive and pervasive bug that will have repercussions for computer users well into the future. Software vendors' reputations depend significantly on their ability to support their products and users once they are in the marketplace.

## COMMENTS AND QUESTIONS

1. Compare the process of writing a computer program to the following activities: writing a novel, writing a poem, writing a symphony, writing a cookbook, preparing a map, designing a building, engineering a bridge, or inventing a new process for manufacturing steel. In what ways is computer programming similar to and different from these creative activities?

2. Is it possible to identify the “most significant” part of a computer program? Is it the idea for the program? The flowchart that structures the program? The source code that implements it? Which of these activities, if any, can be called the “heart” of programming?

#### **D. THE ECONOMICS OF COMPUTER SOFTWARE AND NETWORK MARKETS**

The software industry produces a wide variety of services and products, including operating systems, applications programs, custom programs, embedded systems, and data processing services. Some hardware systems manufacturers, such as Sun Microsystems, develop both operating systems and application software for their systems. There are also many small, independent firms that specialize in aspects of software services and product development. In recent years, a few firms, most notably Microsoft Corporation, have emerged as major forces in the computer software market. As its operating system platforms have become widely adopted, Microsoft has vertically integrated its operations through the development of application programs, and, more recently, electronic commerce product lines. Other software vendors, such as Intuit and Oracle, have concentrated on specific applications program markets, although these companies sometimes extend their product lines as advances in technology open up possibilities for new market niches that may be worth exploiting (e.g., money management software may become useful for managing personal banking as the technology and other developments enable the evolution of this online service). Another major market is for client-server and network management software. Significant players in this market include Novell, Sun Microsystems, and Microsoft.

Software markets evolve rapidly, driven in large part by the rapid pace of technological innovation in computer hardware. New versions of a mass-market software product often appear within a year after the initial release. Indeed, some consumers deliberately delay their purchases of early products and wait for more advanced (and more error-free) versions. Further, the rapid pace of innovation in software products means that the entire structure of the market can change in a very short period of time. At a broader level, new product markets themselves are constantly being created and destroyed as new classes of products become possible and as old software becomes useless or swallowed up into an operating system or another application program.

The rapid pace of software innovation has at least two implications for the law. First, software producers expect obsolescence. They must be prepared either to recover their fixed costs early, before their product is overrun in the marketplace, or to build some sort of long-term, multigeneration relationship with consumers. In either case, they may not need intellectual property protection for their programs beyond the short to medium term (three to ten years). Second, the legal system risks being behind the times unless it evolves rapidly in response to market developments. This is particularly true in antitrust law, where the often-ponderous machinery of antitrust enforcement may not swing into operation until the competitive threat has passed or it is too late to remedy the problem. For similar reasons, intellectual property protection that takes a long time to acquire and enforce tends to provide little effective protection for cutting-edge technologies.

This section discusses two economic concepts of particular relevance to computer software and Internet markets: (1) public goods; and (2) network effects. We then illustrate the operation of the concepts in discussing business strategy and platform competition in the microcomputer industry.

## 1. Public Goods

All markets for products embodying intellectual property exhibit an externality commonly referred to as the “public goods” problem. Public goods have two distinguishing features: (1) nonexcludability (it is difficult to prevent those who do not pay for the good from consuming it); and (2) nonrivalrous competition (additional consumers of the good do not deplete the supply of the good available to others). Military defense is a classic example of a public goods—it benefits all who are defended, whether they pay for the privilege or not. The result of the public goods problem is well-recognized: the private market will undersupply public goods because producers cannot reap the marginal value of their investment in providing such goods.

The computer software market presents an example of this public goods problem. Given how easily and inexpensively computer users may copy computer programs in digital form, vendors have tremendous difficulty excluding non-purchasers from the benefits of innovative computer programs once they are made commercially available. Programs written in object code and stored in ROM can be readily copied by inserting the chip into a laboratory device and downloading the information onto another chip or printing it on paper. Moreover, one person’s use of the information does not detract from any other person’s use of that same information, since the information can be copied without affecting the original. Since the authors and creators of computer software cannot reap the marginal value of their efforts, economic theory suggests that in the absence of other incentives to innovate, they will undersupply technological advances in computer software.

The public goods problem is a familiar one to anyone who has studied intellectual property, since it underlies the economic incentive theory of intellectual property. For a more detailed discussion of public goods in the context of intellectual property, see Peter S. Menell, *Intellectual Property: General Theories*, in *Encyclopedia of Law and Economics* (forthcoming 2000).

## 2. Network Effects

The second potential source of market failure in the computer software market arises from the presence of network externalities. Network externalities exist in markets for products for which the utility or satisfaction that a consumer derives from the product increases with the number of other consumers of the product. The telephone is a classic example of a product for which there are network externalities. The benefits to a person from owning a telephone are a

function of the number of other people owning telephones connected to the same telephone network; the more people on the network, the more people each person can call and receive calls from. Another classic network externality flows from standardization. In this case, the value of learning a particular standard (say, how to use a certain word-processing program) depends on how many other people use that standard. Consider the prevalence of a “normal” typewriter keyboard. Because almost all English language typewriters feature the same keyboard configuration, commonly referred to as “QWERTY,” typists need learn only one keyboard system.

Network externalities also inhere in product standards that allow for the interchangeability of complementary products. Examples of products for which this type of network externality is important are video cassette recorders, CD players, and computer operating systems. As discussed above, general-purpose computer operating systems allow consumers to use a variety of application software programs on the same system-unit hardware. The only requirement is that the application program be coded to work on the operating system embedded in the general computer system. Thus, the operating system serves as a “compatibility nexus” for a particular computer network. Application software producers will develop more programs for systems that are widely used; hardware producers will develop more configurations of disk drives, memory, and other features for popular operating systems. In general, the benefits of a larger computer operating system network include a wider variety of application software that can run on that operating system, lower search costs for consumers seeking particular application programs that run on that operating system, reduced retraining costs, greater labor mobility, and wider availability of compatible hardware configurations and peripherals. See generally Peter S. Menell, *An Analysis of the Scope of Copyright Protection for Application Programs*, 41 *Stan. L. Rev.* 1045 (1989).

Similarly, the greater the standardization of computer-human interfaces for particular application programs, such as word processing or spreadsheets, the easier it will be for computer users to employ their skills in different working environments. Computer users also want the computer-human interface to be as easy to use as possible—that is, “user friendly.” In part, user friendliness is a function of what the user has already learned: things that are familiar tend to be easier to use. More generally, user friendliness is determined by the extent to which the design is tailored to achieve human factor goals—such as maximizing performance speed, minimizing learning time, minimizing rate of errors, and maximizing retention over time.

Standardization can occur by way of market processes, whereby subsequent entrants to a market adopt the standard of an existing firm. This has been referred to as “bandwagon standardization.” See Joseph Farrell, *Standardization and Intellectual Property*, 30 *Jurimetrics J.* 35, 41 (1989). In the computer industry, firms often foster this process by freely licensing the use of a standard and publishing design and interoperability specifications. The UNIX operating system is an example of an open standard promoted by certain private companies, such as Sun Microsystems. It suffers, however, from the lack of



strong sponsorship. Various incompatible versions of UNIX have proliferated over the years, eroding some of the network benefits.

Alternatively, standardization can occur through formal processes, such as government standard-setting and joint development of industry standards by a number of firms. This top-down approach to standard-setting has been common in the telecommunications industry, at least until recently. Regulatory agencies, in consultation with the affected companies, set standards for connecting peripheral devices (including computers) to telephone lines. Only devices that comply with these publicly available standards will operate over the telephone lines.

Finally, industries characterized by strong network externalities will sometimes develop de facto standards, even though neither the companies in the industry nor the government has made any effort to settle on a particular standard or ensure interoperability. The Windows operating system offers a prime example. Microsoft has not “opened” its operating system to competitors. Nonetheless, once the Microsoft operating system reached a certain critical mass of users, the size of its user base began to make it more attractive for other developers to write programs for the Windows platforms. More applications programmers wrote programs to run on Windows because of the large user base; more users adopted the Microsoft system in part because of all the programs being written for it. This positive feedback effect enabled Windows to become the de facto industry standard. In 1999, Windows had close to a 90 percent share of the personal computer operating system market, despite the closed nature of the system and despite the availability of several arguably superior competing systems. For a discussion of these alternative forms of standardization in network markets, see Mark A. Lemley, *Antitrust and the Internet Standardization Problem*, 28 *Conn. L. Rev.* 1041 (1996).

An important economic consideration in markets with significant network externalities is whether firms will have the correct incentives to adopt compatible products, thereby enlarging existing networks. Economists have demonstrated that firms might prefer to adopt noncompatible product standards even though their adoption of compatible products would increase net social welfare. See Michael L. Katz & Carl Shapiro, *Network Externalities, Competition, and Compatibility*, 75 *Am. Econ. Rev.* 424, 435 (May 1985). The explanation for this behavior is that by adopting a compatible standard, a firm enlarges the size of a network that comprises both the adopter’s product and its rivals’ products. This will have the effect of increasing the desirability of the rivals’ products to consumers, thereby reducing the adopter’s market share (although of a larger market) relative to what it would have been had the firm adopted a non-compatible product standard. Although this effect by no means implies that subsequent market entrants will never adopt a compatible operating system or computer-user interface, it does suggest that incentives to adopt compatible standards might be suboptimal.

While a widely adopted product standard can offer important benefits to consumers and firms, it can also “trap” the industry in an obsolete or inferior standard. In essence, the installed base built on the old standard—reflected in

durable goods and human capital (training) specific to the old standard—can create an inertia that makes it much more difficult for any one producer to break away from that standard by introducing a non-compatible product, even if the new standard offers a significant technological improvement over the current standard. See Joseph Farrell & Garth Saloner, *Standardization, Compatibility, and Innovation*, 16 *Rand J. Econ.* 70 (1985). In this way, network externalities can retard innovation and slow or prevent adoption of improved product standards. This problem is particularly significant in industries that develop rapidly, such as the computer industry. Rapid changes make the adoption of standards (formal or informal) a hazardous enterprise.

A frequently cited example of standard stagnation is the persistence of the standard QWERTY typewriter keyboard despite the availability of a better key configuration. United States Navy studies and typing speed world records indicated that the Dvorak Simplex Keyboard, a typewriter key configuration patented in 1932 by August Dvorak and W.L. Dealey, was significantly more efficient than the QWERTY system. Adoption of the better standard may have been effectively stymied by “switching costs”—the costs of converting or replacing QWERTY keyboards and retraining those who use this system.<sup>7</sup> Because of the fear that national standards would exacerbate the inertia problem, the National Bureau of Standards declined to set interface standards for computers in the early 1970s. Subsequent developments proved that a wise decision.

Nonetheless, standards are essential in many contexts. While the U.S. government did not adopt computer interface standards in the 1970s, the hardware industry had to develop *de facto* standards in order to allow computers and peripheral devices to communicate with each other. The software industry faced similar problems in ensuring compatibility between operating systems and applications programs in the 1980s, and the network industry faces the same problem today. In each case, the problem goes away if one company controls the market for both goods, as AT&T did in the telephone industry until 1984. If AT&T is the only company to sell both telephone services and telephones, there is unlikely to be a problem in ensuring that your telephone works with your phone line. But this form of standardization comes at the expense of competition. Controlling standards and compatibility in dynamic industries is one of the most vexing policy problems in the computer and telecommunications industries. Yet sometimes advances in technology can aid in solving these problems.

The economics of network markets suggests a range of business strategies that take advantage of network effects. See generally Carl Shapiro & Hal R. Varian, *Information Rules* (1999). Network markets are inherently “tippy.” Given the large payoff to becoming a *de facto* industry standard, it behooves firms to invest heavily in strategies to attract adopters. Once these customers make investments in installing a product, learning its attributes, and developing data bases, they become locked into the product. This affords the product

7. This story has been challenged on its factual merits, however. See S.J. Liebowitz & Stephen E. Margolis, *The Fable of the Keys*, 33 *J.L. & Econ.* 1 (1990).

sponsor substantial latitude to charge higher prices, bundle the platform with other products, and exercise some degree of control over innovation and competition in related markets. The history of the microcomputer illustrates the importance of understanding these lessons and the challenges that network markets pose for policymakers, judges, business decision-makers, and lawyers. For a comprehensive discussion of the legal significance of network effects, see Mark A. Lemley & David McGowan, *Legal Implications of Network Economic Effects*, 86 Cal. L. Rev. 479 (1998).

### **3. Platform Competition in the Computer Industry**

Early entrants into the microcomputer market followed the “single vendor” model of the mainframe and minicomputer markets. Osborne, Apple, Kaypro, and others developed operating systems unique to their computers, which they packaged with application software and marketed as complete systems. These computer systems were not compatible. This enabled these vendors to control markets for software and peripheral equipment for their computer systems, but it had the disadvantage of segmenting the market for application programs.

Similarly, IBM declined to license the leading operating system for Intel 8080 chips, Digital Research Incorporated’s CP/M (control program/monitor) operating system. Instead, IBM licensed a new operating system (DOS) from Microsoft Corporation, a young company that had attracted some attention for having developed a version of the BASIC computer language for early microcomputers. Even IBM’s use of Microsoft’s DOS platform, however, was not completely open to the industry because IBM refused to license its BIOS chip. Due to IBM’s strong trademark and marketing network, this decision led to the rapid demise of the CP/M platform and hurt those OEMs that had built their microcomputers upon the CP/M platform.

Building on the name recognition flowing from its association with the IBM PC, Microsoft selected a business model aimed at developing and promoting relationships with a broad range of microcomputer companies, the OEM and ISVs. This model encouraged compatibility of hardware and software products in the industry and enabled software vendors to reach a much larger market for their products: the many computer systems operating on the MS-DOS platform.

As its operating system market grew, Microsoft appreciated the fundamental importance of compatibility in microcomputer markets. In the words of one of its senior executives,

[c]onsumers value compatibility. Compatibility makes computers easier to learn and leads to a plethora of applications. Microsoft understood in the early 1980s that the potential of personal computers would not be unleashed until the industry provided a broad set of compatible hardware and software products. To help bring

that about, Microsoft developed and promoted an important part of the standard personal computer architecture—a set of operating systems.<sup>8</sup>

Microsoft's business strategy came to encompass four principal elements all directed toward promoting the widespread adoption of the Microsoft operating system platform: (a) encouraging the development of application programs to run on an evolving Microsoft platform; (b) building an intellectual property portfolio to protect its operating system platform; (c) developing long-term relationships with OEMs to promote Microsoft's operating system platforms while preventing direct competition in the platform market and discouraging adoption of competing operating systems; and (d) evolving the operating system platform in a way that was backward-compatible (so as not to strand the existing base of computer users who had developed programs and data bases for prior operating system) while incorporating enhanced functionality made possible by advances in microprocessor, other hardware, and software technology. In particular, Microsoft began developing a graphical user interface OS and various network support products (LAN and Windows NT).

Microsoft understood well the lessons of network economics. It successfully registered its trademarks and invested heavily in advertising them and in expanding its network. The company used pricing, contractual, and bundling strategies to lock consumers into its products while discouraging adoption of competing products. For example, it offered its operating system to OEMs at relatively low prices in contracts that required the OEMs to pay a royalty for every system sold, whether or not they actually installed a Microsoft operating system. Because many microcomputer buyers preferred the Microsoft platform in order to run the large stock of software that was available, OEMs had little incentive to offer competing operating systems. These "per processor" agreements made the marginal cost of installing Microsoft's operating system zero. In the 1990s, Microsoft successfully leveraged its market power into other markets as well. It bundled its application programs for office use (Word, Excel, Access, Powerpoint) into its Office suite. As the World Wide Web became popular, Microsoft integrated its Internet browser (Microsoft Explorer) directly into Windows. These practices took effective advantage of the economics of network markets and expanded Microsoft's dominance throughout the software domain.

Although Microsoft's dominance of the microcomputer industry remains strong, technological developments continue to reshape the competitive landscape. In the mid-1990s, Sun Microsystems introduced Java, which Sun describes as "a standardized application programming environment that affords software developers the opportunity to create and distribute a single version of programming code that is capable of operating on many different, otherwise incompatible systems platforms." See Mark A. Lemley & David McGowan, *Could Java Change Everything? The Competitive Propriety of a Proprietary Standard*, 43 *Antitrust Bull.* 715 (1998). Thus described, Java offers the promise of platform-independent computing, the equivalent of a standardized "meta"-

8. Direct testimony of Paul Maritz, group vice president for platforms and applications, ¶127, *United States v. Microsoft Corp.*, No. 98-1232, U.S.D.C. DC (public version).

operating system that would allow both consumers and programmers to choose operating systems free of the constraints of network effects. Java may reduce the opportunity cost incurred by those using platforms other than the market standard, thereby facilitating transition among platforms and alleviating concerns that Microsoft can exploit its installed base of users and programmers.

A “standardized application programming environment” sounds to many like an operating system: a good for which applications may be written and that coordinates the computer’s functions. The difference is in the meta-operating system characteristic Sun ascribes to Java technology—it is billed as the operating system whose job it is to allow any operating system to function with applications written for any other operating system (or possibly without any system other than Java in mind), so long as both the operating system and the application are themselves compatible with Java. Java may represent the realization of the long-sought platform-independent competitive horizon. Ironically, this platform-independent world would be brought to us by a proprietary program to which Sun owns the intellectual property rights.

Sun faced the problem all companies who want to compete in a network market must deal with: its Java product is new and therefore lacks an installed base. Java has achieved rapid, widespread, and growing acceptance in the software community. Hundreds of thousands of programmers are writing Java code, and some companies are even designing hardware products optimized for Java. Like Microsoft, Sun has sought to follow the lessons of network economics by promoting adoption of its product through advertising, education, competitive pricing, and widespread licensing.

In addition to the Java platform, a grassroots effort among programmers has attracted attention to the open source Linux operating system. Linux was developed from the UNIX operating system, an operating system platform initially developed by AT&T that became fragmented as many companies developed different, incompatible versions. What distinguishes the Linux code from the many other versions of UNIX is that its principal developer and chief promoter, Linus Torvalds, has committed to maintaining Linux as an open standard. Linux has developed a loyal following within the academic world. The size of its network has grown sufficiently large that some commercial software vendors have begun writing programs that run on the Linux platform. At this writing, it remains to be seen whether either Java or Linux will offer significant competition to the Microsoft operating system.

## COMMENTS AND QUESTIONS

1. Does intellectual property protection for products featuring network externalities increase or decrease the likelihood that firms will seek to develop proprietary standards? For an argument that the problem of market power is exacerbated by the availability of intellectual property protection for many of the potential standards in a network market, see Mark A. Lemley & David McGowan, *Could Java Change Everything? The Competitive Propriety of a Proprietary Standard*, 43 *Antitrust Bull.* 715 (1998).

2. How should the presence of network externalities with regard to computer software affect the analysis of intellectual property protection? Of what relevance are network externalities to the scope of copyright, patent, or trademark protection? Should the network externalities of operating systems be treated in the same manner as network externalities of user interfaces? See Peter S. Menell, *An Epitaph for Traditional Copyright Protection of Network Features of Computer Software*, 43 *Antitrust Bull.* 651 (1998).

3. The development of Java in particular blurs the line between computer operating systems and Internet-related applications programs. In so doing, it raises the possibility of private ownership of the protocols that govern the Internet. Right now, the Internet is built on a fundamentally open architecture, as Jane Winn explains:

The National Research Council has noted that the Internet is now “open” in at least the following four senses. It is open to users because it does not force users into closed groups or deny access to any sectors of society but instead permits universal connectivity, like the telephone system. It is open to service providers because it provides an open and accessible environment for competing commercial and intellectual interests. For example, competitive access for information providers is permitted. It is open to network providers because any network provider can meet the necessary requirements to attach and become a part of the aggregate of interconnected networks. It is open to change because it continually permits the introduction to new applications and services. It is not limited to only one application, as in television. It also permits the introduction of new technologies as they become available.

Jane Kaufman Winn, *Open Systems, Free Markets, and Regulation of Commerce*, 72 *Tul. L. Rev.* 1177 (1998). It is certainly possible, however, to imagine part or all of the Internet protocols being proprietary. Suppose Microsoft develops a privately owned extension to HTML, the basic markup language used to encode information on the Web. If that extension were to become a standard among Web developers, Microsoft would control access to the Internet—at least for anyone who wanted to use that protocol.

We discuss the long-running debate between open and closed systems in several places throughout this book.

## **E. OVERVIEW OF INTELLECTUAL PROPERTY PROTECTION FOR COMPUTER SOFTWARE**

Information technology industries rely heavily on intellectual property law to protect their products and other assets. For this reason, this book provides considerable coverage of intellectual property issues. Chapter 1 demonstrates how trade secrecy law can be used to protect some valuable aspects of software and other information technologies. Chapter 2 considers the extent to which copyright law provides protection to computer programs. Chapter 3 traces the

evolution of patent protection for software innovations. Chapter 4 explores the supplemental protection that may be available to information technology firms from trademark law. In addition, Chapter 5 considers some *sui generis* forms of legal protection that have been considered or adopted to protect certain information technology products (e.g., semiconductor chip designs and databases) for which existing intellectual property regimes may provide inadequate protection. Chapter 8 also provides an overview of some differences in the intellectual property rules applicable to computer software and other information technology innovations in the international market.

While computer software may have been the first digital information product to rely crucially on intellectual property laws, other digital information products rely heavily on intellectual property protection as well. Hence, in Part II, Chapters 10 and 11 will consider a range of digital intellectual property issues that arise in the context of the Internet and the World Wide Web.

Because so much of this book emphasizes intellectual property law and because information technology developers need to develop complex strategies built up as multiple forms of legal protection, this chapter provides a brief overview of the specific forms of intellectual property law. This may be helpful before the reader delves into the book's detailed consideration of how each mode of protection may be used to protect some aspects of software or other information technology products or services.

Intellectual property law comprises a diverse array of state and federal legal regimes. Under state law, intellectual work can be protected through contract law, trade secrecy protection, and trademark protection. Under federal law, intellectual work can be protected by patent law, copyright law, and trademark/trade dress protection under the Lanham Act. These regimes interact in a number of ways, and the subject matter of these various modes of legal protection overlap. For example, both trade secret law and patent protection cover technological advances. Although federal intellectual property protection has primacy, federal law does not preempt state trade secret law, which leaves both bodies of law to operate in tandem.<sup>9</sup> By contrast, copyright protection does not extend to "any idea, procedure, process, system, method of operation, concept, principle, or discovery," so as not to undermine the role of the patent law (which erects a higher threshold for the protection of functional works). 17 U.S.C. §102(b).

## 1. Trade Secret Law

Software developers and other information technology firms often rely on trade secret law to protect valuable information. The source code forms of

9. See *Kewanee Oil Co. v. Bicron Corp.*, 416 U.S. 470 (1974) (patent law does not preempt trade secret law); H.R. Rep. No. 1476, 94th Cong., 2d Sess. 132 (1976) (legislative history of §301 of the Copyright Act stating that trade secret law would remain unaffected by the Act so long as it contains elements different from copyright infringement); *Warrington Assocs. v. Real-Time Eng'g Sys.*, 522 F. Supp. 367, 368-369 (N.D. Ill. 1981) (trade secret not preempted by Copyright Act).

programs and other internal design documentation, for example, are often locked up in secure places and held as trade secrets. Trade secret law protects against the misappropriation of such commercially valuable secret information by unfair or commercially unreasonable means, whether done by competitors or anyone else.

Every state protects trade secrets through either common law or statutory regimes. In 1979, the National Conference of Commissioners on Uniform State Laws promulgated the Uniform Trade Secrets Act (UTSA), which has since been enacted by 40 states and the District of Columbia. The Act prohibits the actual or threatened misappropriation of a trade secret. The UTSA defines a “trade secret” broadly as

information, including a formula, pattern, compilation, program, device, method, technique, or process, that:

- (i) derives independent economic value, actual or potential, from not being generally known to, and not being readily ascertainable by proper means by, other persons who can obtain economic value from its disclosure or use, and
- (ii) is the subject of efforts that are reasonable under the circumstances to maintain its secrecy.

U.T.S.A. §4. Trade secrets have potentially unlimited duration. As the law’s name suggests, information must be a secret if it is to be protected by this law. However, only relative and not absolute secrecy is required. The owner of a trade secret must take reasonable steps to maintain this secrecy. Although trade secrets are often licensed to other firms, the “reasonable steps to maintain secrecy” requirement can generally be satisfied by provisions in the license agreement requiring the licensee to limit access to and use of the trade secret information. In addition, licensees will be under a nondisclosure obligation if the circumstances of the disclosure of a secret give rise to an understanding that the information must be carefully protected. Trade secrets have no definite term of protection, but they will be protected only as long as they are secret. Once a trade secret is disclosed, the legal protection dissipates.

Courts will generally find misappropriation of trade secrets in two circumstances: (1) when a person uses theft or other improper means (e.g., bribery of employees) to acquire the secrets, or (2) where someone uses or discloses the secrets in violation of a confidential relationship. Trade secret laws do not protect against independent discovery or invention of the same information. Nor do they prevent competitors from reverse engineering a lawfully obtained product in order to discern the secrets it might contain. Successful plaintiffs in trade secret cases are entitled to recover damages attributable to the misappropriation of the secret and in some cases to an injunction against use or further disclosure of the secret. However, injunctions may be limited in duration to the time it would have taken a competitor to independently develop the secret or otherwise discover it by fair means.



## 2. Copyright Law

Software developers and other information technology firms also routinely rely on copyright law. Copyright law protects object code versions of computer programs against exact copying. This law also provides significant protection to other aspects of programs—for example, nonfunctional design elements of user interfaces. There is some debate, as Chapter 2 illustrates, about how far copyright protection extends to the structure and organization of programs.

Unlike trade secrecy law, copyright protection is governed exclusively by federal law, and infringement suits may only be brought in federal courts in the United States. The Constitution provides Congress with the power to “promote the progress of science and [the] useful arts, by securing for limited times to authors . . . the exclusive right in their . . . writings.” Congress has used this power to provide protection to “original works of authorship” that have been “fixed in a tangible medium of expression,” including computer programs, photographs, paintings, sculptures, musical compositions, dramatic plays, and sound recordings. To qualify for copyright protection, a work must be “original” in the sense that it represents an author’s intellectual creation, but it need not be particularly inventive. Works made by employees in the course of their employment are deemed the work of the employer for purposes of copyright ownership.

Copyrights exist from the moment that a work is fixed in a tangible medium of expression. Until legislation was passed in 1998 extending the duration of copyrights by 20 years, protection existed upon the creation of the work and lasted for the life of the author plus 50 years, or 75 years in the case of entity authors. 17 U.S.C. §302. The owner of a copyright has the exclusive right to make copies of his work, prepare derivative works, distribute his works, and perform and display his works in public. 17 U.S.C. §106.

The “useful article” doctrine and the idea/expression dichotomy limit the scope of copyright law so as not to undermine the patent law’s protection of functional inventions. The useful article doctrine allows copyright protection for expressive features of “pictorial, graphic, and sculptural works” only to the extent that such features “can be identified separately from, and are capable of existing independently of, the utilitarian aspects of the article.” 17 U.S.C. §101 (definition of pictorial, graphic, and sculptural works). The idea/expression dichotomy excludes from copyright protection

any idea, procedure, process, system, method of operation, concept, principle, or discovery regardless of the form in which it is described, explained, illustrated, or embodied in such work.

17 U.S.C. §102(b). In *Baker v. Selden*, 101 U.S. 99 (1879), the holder of a copyright in a book describing a particular system of bookkeeping sought to prevent others from using that method (and the forms described). The Supreme Court held that methods and forms needed to use such methods are not

copyrightable. The Court enunciated the idea/expression limitation on copyright protection:

To give to the author of the book an exclusive property in the art described therein, when no examination of its novelty has ever been officially made, would be a surprise and a fraud upon the public. That is the province of letters-patent, not of copyright. The claim to an invention or discovery of an art or manufacture must be subjected to the examination of the Patent Office before an exclusive right therein can be obtained; and it can only be secured by a patent from the government.

101 U.S. at 104.

The fundamental purpose underlying U.S. copyright law is to promote the creation and dissemination of knowledge. Some authors and artists may be unwilling to invest the time and effort involved in producing a new work of authorship if they know that others can simply copy their work. The cheaper and easier it is to make multiple copies, the greater the potential market failure. Copyright law ameliorates this problem by allowing the copyright owner to capture a significant part of the commercial value (if any) of her work for a limited period of time. One way that copyright encourages the dissemination of new works of authorship is to enable subsequent authors and artists to “build on” the works of their predecessors, either by using the unprotectable ideas conveyed in existing works or by drawing upon works in the public domain (whether because the copyright term has expired, the work is ineligible for copyright protection, or the author dedicated the work to the public domain).

Although not required to obtain copyright protection, software developers may register a claim of copyright with the U.S. Copyright Office, and there are some advantages to doing so. A registration certificate generally constitutes prima facie evidence of the validity of the copyright and of the claim of the registrant to be the owner of the copyright. Second, prompt registration is necessary to qualify for an award of attorney fees or statutory damages under U.S. law. Third, a registration certificate is a necessary precondition for U.S. authors to bring infringement actions in U.S. courts. However, registration is not a precondition for a work to be protected by copyright law. Instead, copyright subsists automatically in original works of authorship from the time they are first created. The duration of a copyright now extends for the life of an individual author plus 70 years (or 95 years from first publication when works are produced by corporations).

Unlike patent law, copyright law protects against only the copying of works, and not making, using, or selling works. Hence, copyright law does not prevent others who independently develop a work from competing with the first inventor. In addition, copyright protection is moderated by the fair use doctrine, which allows others to make limited use of a work for teaching, research, and other purposes. 17 U.S.C. §107.

### **3. Patent Law**

Patents have long been used to protect computer hardware systems and are increasingly used to protect software innovations, such as algorithms and data structures, as well as methods of doing business in the electronic environment. Chapters 3 and 8 show that the United States and other nations differ in their approaches to patent protections for computer software.

Patents are granted to inventors who meet several substantive and procedural requirements. First, the invention must fall within the class of patentable subject matter. Generally, this means it cannot be a purely mental discovery or a law of nature. Second, the resulting product or process must be useful. It must also be novel and be more than an obvious advance over the prior art. A patent is awarded in the United States to the first to invent, not the first to file, a patent claim, as is common in other countries. A patent gives the patentee the exclusive right to use or license the patented product or process in the United States for 20 years from the date the patent application was filed. A patent is not automatically awarded. Rather, the Patent Office requires its examiners to carefully scrutinize the application and make a judgment about whether the patent should be issued. Many applications are disapproved, and others are approved only after significant narrowing of the patent claims.

If the Patent Office grants an inventor a patent, the patent is published, so that everyone else can learn what the inventor has made and how to make it. The inventor gets the right to prevent anyone else in the United States from making, using, or selling a product or process covered by the “claims” of the patent (the portion of the patent defining the invention) while it is in force. If another party does make, use, or sell the patented invention during the term of this exclusive right, the patent owner (called the patentee) is entitled to sue them for infringement of the patent. If the patentee proves that the defendant is in fact infringing the patented invention, the patentee can obtain an injunction against the infringer and damages for past infringement. A patentee can also prevent others from importing infringing works into the United States.

Under 35 U.S.C. §§171-173, an inventor may obtain a design patent for “any new, original and ornamental design for an article of manufacture.” In order to receive protection, the inventor must establish that the design is novel, nonobvious, and ornamental, which means reflecting aesthetic skill and artistic conception. If a design is primarily functional rather than ornamental or is dictated by functional considerations, then it is not eligible for a design patent. Design patents are valid for a term of 14 years from the date of issuance.

### **4. Trademark Law**

Trademark law protects words, symbols, and other indicia that serve to identify the source of goods or services against confusingly similar uses in

commerce. Examples of protectable trademarks include corporate and product names (e.g., Apple Computer, Inc., and iMac as a name of a computer made by Apple), logos (e.g., the cube-like shape that signifies Sun Microsystem's products), slogans (e.g., "Intel inside"), and product configurations (e.g., nonfunctional aspects of computer keyboard layouts) as long as the significance of the symbol lies principally in designating the origin of goods or services. The word, symbol, or indicia cannot be merely a description of the good itself, or a generic term for a class of goods or services.<sup>10</sup> Further, the identifying mark must not be a functional element of the product itself, but must principally serve an identifying purpose. Functionality in the context of trademark law means that there is no competitive necessity for other market participants to use the same or a very similar designation in order to compete effectively in the market. In addition, trademark owners forfeit their rights by licensing their mark to others without controlling and monitoring the nature and quality of the goods or services with which the mark is used. Finally, trademark protection is directly tied to the use of the mark to identify goods or services in commerce. Under U.S. law, trademarks do not expire on any particular date; they continue in force until "abandoned" by their owner. In some countries, however, as Chapter 8 explains, trademark rights arise by virtue of registration and may expire unless renewed in a timely fashion.

While state common law and statutory law may offer significant protection to trademarks in the United States, the most significant source of trademark protection on a national basis is that conferred under the federal trademark law known as the Lanham Act, 15 U.S.C. §§1051 et seq. This Act establishes a procedure by which federal officials examine trademark registration applications and issue trademark registrations that confer significant benefits upon the registrants. However, federal trademark registration is not necessary to obtain trademark protection. A trademark owner who believes that another is using the same or a similar mark to identify competing goods can bring suit for trademark infringement under state law. The ultimate question in trademark litigation is whether consumers are likely to be confused if two different firms are using the same or similar marks in the marketplace. If so, the trademark owner is entitled to an injunction against the confusing use, damages for past infringement, and in some cases the seizure and destruction of infringing goods.

In recent years, the Lanham Act has been amended to provide federal protection to highly distinctive marks (generally those that are famous) against "dilution" of the good will associated with the mark. A number of state trademark dilution laws also exist. Trademark dilution laws protect trademarks against uses of trademarks even when consumers are unlikely to be confused by the coexistence of the marks in commerce. Dilution laws protect against uses that would cause a blurring of the distinctiveness of the mark (e.g., IBM for

10. "Genericide" occurs when consumers associate a trademark with a general product category rather than a designation of source. Prominent examples of trademarks that became generic include thermos, escalator, and linoleum. The "x86" designation has been deemed generic as a way of denoting microprocessors, which explains in part why the latest version of the Intel microprocessor was dubbed "Pentium" rather than "586."

automobiles) or tarnishment (e.g., Lotus as a name for a pornographic bookstore).

## **5. Sui Generis Laws**

Proposals for sui generis intellectual property legislation for software and other information technology products have arisen periodically in legal and policy circles. Generally, the rationale is that existing intellectual property systems did not provide sufficient protection to promote adequate levels of investment in the development of this technology. Sometimes the argument is reversed—that existing intellectual property statutes provide too much protection, or at least the wrong kind of protection.

Although sui generis proposals for software have not taken hold in the United States, this country has adopted a sui generis form of legal protection for semiconductor chip designs. As this book goes to press, the United States is also considering adoption of a sui generis form of legal protection for collections of information that may not meet copyright standards. The European Union has already enacted such a law. Chapter 5 will discuss these laws and proposals.

## **6. Contract Law**

Traditional contract law provides an important framework for protecting the intellectual work embodied in software products. Contract law establishes the rules for enforcing the bargains of contracting parties. It can be used to structure a wide range of software transactions and relationships, including service contracts for the design and implementation of software systems or programs on a client's computer, sales of software products, and lease agreements. In general, the Uniform Commercial Code governs contracts for goods, while common law and other state statutes govern service contracts. Contracts governing the disclosure of information, rights to use technology, and rights to make copies of a work fall into the latter category. At this writing, a proposed statute (the Uniform Computer Information Transactions Act, or UCITA) would govern transactions in "computer information" and would fundamentally expand the role of contract in protecting software and digital information.

