Appendix B: Introduction to Computer Technology

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 eight per cent 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 make 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 appendix 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.

A brief note on methodology in this appendix 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. We reference a number of excellent sources providing detailed background in our summaries below. 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. Rather, more recent developments (including the growth of the Internet) are discussed in the sections that follow, and in chapter VII of the main text.

Following the invention of the abacus approximately 5000 years ago, the field of computing machines did not develop significantly until the 18th 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 19th 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-19th 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 19th century, a U.S. Census Bureau agent named Herman Hollerith developed a punched-card tabulating machine to automate the census. Drawing upon the use of "punched photography" by railroads (to encript passengers' hair and eye color on tickets), Hollerith proposed the encoding of census data for each person on a separate card which 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 1920's, 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, eventually developing 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. They 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 electromechanical 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) 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. 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 out-sourcing 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 $1000) 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, singe-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 independent software vendors (ISVs) and hardware manufacturers developed and marketed software and peripheral products to run on the IBM PC. IBM actively encouraged independent software vendors and the makers of peripheral equipment (e.g., printers, 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)[1] for transferring data within the PC that hindered other 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 independent software vendors (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.

---

[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.

Microsoft and Intel retained rights to market their products to other OEMs (original equipment manufacturers) 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 in the marketplace. 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. 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 that 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.

As the industry developed and 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 governmental 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 on a fee basis. This early software industry offered specialized services on a contract basis. They 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 flow chart 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% 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 governmental

agencies. During this same period, CSC increasingly shifted its software 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 1000 software firms in the United States offering more than 3000 products.

The proliferation of minicomputers in the early 1970s fostered the growth of independent software vendors and the shift away from custom programming and support services toward pre-packaged software products. The introduction of the microcomputer in the mid- to late 1970s dramatically changed the software industry. With relatively small investments, computer programmers could develop software for the growing numbers of microcomputer users. Beginning in the late 1970s, independent software vendors (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% 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 spreadsheeet 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. <u>An Introduction to Computer Technology</u>

Virtually unknown 50 years ago, computers literally surround most Americans in their daily lives today. In their most easily recognized form, mainframe and mini-

computers 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 are known as computer programs, and they will be 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. ROM chips have 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" which 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 upon 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 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 which 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 videogame 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 semiconductor 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" 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, 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 may order the deletion of a file, but generally, the operating system will actually carry 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 (of which more in Chapter VII of the main text).

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 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 subsection 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 1990's. [We discuss this suit in more detail in Chapter VIII of the main text].

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 a 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 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 in the 1990s that the Internet became 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 per cent 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 times and cost of transferring information regardless of 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 the two computers. If the computers are physically separated by large 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 kind of hardware to enable telephone transmissions of digital data is known as 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. Local area networks provide all of the communications advantages described above -- they enable employees to communicate by e-mail, to send files to one another, and 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 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 that rely on large information databases which must be regularly updated (for example, sales companies, airlines, hotels). 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, and not by 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 by dozens or hundreds of computers, many 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 require136 separate communication lines (17 computers times 16 divided by 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 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 APRANET successfully implemented packet switching technology in a four-node network. The ARPANET grew to more than 20 sites by 1971 and over 200 sites by 1981. The ARPANET paved the way for the Internet, an international collection of *inter*connected computer *net*works 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 that 9 million host computers were connected to the Internet and it is projected that this number will exceed 200 million by the 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, 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 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 standalone 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 local area networks (LANs) and over the Internet. And more and more programs are distributed in pieces on an as-needed basis over the Internet. 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.

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. We focus here upon the methods used in the development of relatively complex commercial operating systems and application programs, although many of the stages can be found in other programming contexts.

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 the relatively simple operation of a clock to the highly complex control 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 computer more effective and easy 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, and J.H. Reichman, *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 Colum. L Rev. 2308, 2326-32 (1994). They argue:

> 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 works: 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 (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 tasks 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."2 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 multi-stage process often analogized to a waterfall.[3] 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 (3rd ed. 1998); Grady Booch, Object-Oriented Analysis and Design with Applications (2nd ed. 1994); Ian Somerville, 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. This stage of the process identifies 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 its capability.

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

---

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." Samuelson, Davis, Kapor, & Reichman, *supra*, 94 **COLUM. L. REV.** at 2322.

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

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 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 invetnion, 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.

Pamela Samuelson, Randall Davis, Mitchell D. Kapor, and J.H. Reichman, *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 Colum. L Rev. 2308, 2326-32 (1994).

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 comprises 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 data necessary, desired functionality, and hardware and security constraints. They might develop tables or flow charts 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 – e.g., billing clients, filing documents, ordering supplies, preparing budgets, 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 will map data inputs and outputs and assess 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 "flow chart" 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 B-1.

[insert Figure B-1 here. Figure B-1 is identical to IPNTA 1st ed Figure 7-4 on page 839]

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 change 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 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 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 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 draw 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 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 sub-classes. 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 affecting 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.[4]

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.[5]

This 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 traditional top down approaches. The adaptability of object-oriented software programs substantially reduces the risk that a large investment in software design will 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 and David W. O'Brien, *Encouraging Software Reuse*, 49 **Stan. L. Rev.** 255, 259-68 (1997). This reuse of "chunks" of computer code is not only valuable because it saves the cost of rewriting program 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 across firms may enhance software compatibility, decrease programming costs, and even improve the safety and reliability of software (since mission-critical software will not be written from scratch for each new product, but can use code that has already been tested and proven to work).

---

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

[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).

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 (3rd 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 which aims to understand how human beings process information so that products can better be 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, 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.[6]

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.

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) 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 B-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.

---

[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).

[insert Figure B-2 here. Figure B-2 is identical to IPNTA 1st ed Figure 7-1 on page 835]

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 is important to the treatment of software under copyright law.

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 accomplishing 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 *b*inary dig*it* 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-1980's, most computers acted upon words of eight bits, known as a "byte" (e.g., "01101001"). More recently computers have been designed to act upon words of 16, 32, or 64 bits. Figure B-3 contains 24 of 1,674 lines of the object code derived from the PASCAL program in Figure B-2.

[insert Figure B-3 here. Figure B-3 is identical to IPNTA 1st ed Figure 7-2 on page 836]

Obviously, high-level programming languages are more compact than machine language, as well as being easier for humans to decipher. This results from the fact that the representation of complex information as either a 1 or a 0 requires an enormous number of 1s and 0s.

Because of the difficulty of using and deciphering object code, programmers developed an intermediate level language referred to as "assembly language" to facilitate the translation of higher level languages to object code. Assembly language uses abbreviated alphanumeric symbols such as "ADC", which means "add with carry." For older models of Apple computers, "ADC" translates to "01101001" in object code. An assembly program or assembler translates assembly language into object code. Figure B-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.

[insert Figure B-4 here. Figure B-4 is identical to IPNTA 1st ed Figure 7-3 on page 838]

5. Software Validation and Maintenance

Although this discussion may suggest that programming is a linear process, in fact, it is 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 which 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 more generally, the software design team will run 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 conduct alpha (or 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 engage a further level of testing commonly referred to a 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 until the product has been in use for a long time. The Year 2000 (Y2k) problem is to a large extent a massive and pervasive bug that will have repercussions for computer users well into the future. Software vendors’

reputation 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, inventing a new process for manufacturing steel. In what ways is computer programming similar and different from these creative activities?

2. Is it possible to identify the "most significant" part of 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?