Chapter 4

Software Technology and the Law

Contents

Boxes

Figures

Chapter 4
Software Technology and the Law

Introduction

There are intellectual property issues associated with four elements of a program: the program function, the external design, the user interface design, and the program code. The first section of this chapter describes the technology behind each of these elements. The second section outlines the application of existing intellectual property laws to each element, and discusses the policy issues associated with the current level of protection. There have been various policy positions advanced for maintaining or changing the scope of protection, and most of these policy positions have been supported by legal arguments; the final section of the chapter briefly summarizes these legal arguments.

Technology

Introduction

One way to think about computers and programs[1] is to look at the hardware. The core parts of the computer are the processor and memory. Both the processor and memory usually consist of one or more *integrated circuits,* which are semiconductor chips that manipulate digital electronic information. The processor and memory work together to perform logical and arithmetic operations on data; the program is stored in memory and specifies the order in which the operations are to be performed.

A program consists of a list of instructions. Each type of processor has an instruction set—a set of operations that it is capable of performing. Most of these operations are simple; for example, a typical instruction set would include an instruction for operations such as moving data from memory to the processor, logical operations such as checking if two pieces of data have the same value, and arithmetic operations such as adding two numbers. A program is executed when the instructions are transferred to the processor and the processor performs the specified operations.

Inside the computer, ‘‘instructions’ and ‘‘data’ arc both patterns of electronic signals. These signals can take one of two values; to make it easier to comprehend what is happening inside the computer, programmers represent one of the values with the symbol” 1,’ and the other with the symbol ‘‘ O.” For example, the addition instruction for the processor that is used in most microcomputers may be represented as “00000100.” In the same way, in most computers the letter “A’ is represented by the pattern of signals corresponding to “01000001. ” The complex functions that programmed computers perform for users often seem far removed from the patterns of electronic signals and very simple operations that characterize the computer at the hardware level. Computers perform complex tasks by performing a large number of simple operations in sequence—typically millions of operations per second.

The processor and memory are usually part of a larger system. Data to be used in a computation may be read from a disk or tape drive. Data can also be exchanged or shared with other computers using a network. The data is exchanged using *communications protocols,* which specify the format and sequence in which data is sent. It is also possible for part of a computation to be carried out on a distant computer. Sometimes specialized computers are used for different parts of a task; for example, a supercomputer may carry out the computationally intensive portions of a task, while a workstation is used for displaying the results of the computation.

There are also a variety of input and output devices for communicating with the user. The display is the output device most commonly used for providing users with information about the results of a computation. More advanced displays, faster processors, and cheaper memory enable program developers to go beyond the simple display of text to include graphics. Color monitors are also increasingly common. In some applications, small displays inside helmets or goggles are used to give users the

[1] For an introduction to computers and computer science, see Alan W. Biermann, *Great Ideas in Computer Science* (Cambridge, MA: MIT Press, 1990).

illusion of a three-dimensional image.[2] Sound, which can include warning tones, music, and synthesized speech, may also be used to provide information to users.

Input devices include the keyboard, for entering text, and the *mouse,* a pointer used for drawing figures or selecting commands on a screen. The use of a special pen can also allow the entry of written information and commands.[3] Research is under way on speech recognition technologies that allow commands to be spoken instead of typed or selected using a mouse. Other experimental input devices detect eye movements[4] or gestures made with a special glove.

Sometimes the memory and the processor are not part of a conventional computer, but are *embedded in* industrial machinery and other devices. The processor receives information from a variety of sources, processes the data, and uses the results of the computation to direct the operation of a machine. Examples of embedded systems are the microprocessor-based controllers used in appliances, automobiles, and industrial processes.

Program Function

Programmed computers perform a series of calculations to transform input values to output values. A well-defined computational procedure for taking an input and producing an output is called an *algorithm.*[5] Algorithms are tools for solving computational problems. The statement of a problem specifies in general terms the desired relationship between the input and the output, while the algorithm describes a specific computational procedure for achieving the input/output relationship.[6]

The transformation of input data to output data can also be performed in hardware. Integrated circuits can perform the same simple logical and arithmetic operations that programmed computers perform. Connecting together these electronic circuits has the same effect as programming a computer. Just as the calculation that the computer will perform can be understood by looking at the program, the calculation that a circuit will perform can be understood by looking at the circuit diagram.

The choice of whether to perform the calculation by programming a computer or building a circuit is an engineering decision. Often, a calculation can be performed more quickly by hardware, which may be an important consideration in some applications such as signal processing. On the other hand, programming a computer is potentially less costly and more flexible. The function of a programmed computer can be changed by writing a new program; with hardware, however, a new circuit must be built.

Often many different problems can be modeled in a similar way, and solved using the same class of algorithms. For example, many applications that operate on speech signals and video images use similar signal processing algorithms. Searching and sorting algorithms are also among the basic tools that are commonly used in software development. Problems such as finding the fastest route between two cities or determining when to perform tasks in a manufacturing process may be modeled in a way that makes them solvable by using graph algorithms.

External Design

Programs have an external design or interface—the conventions for communication between the program and the user or other programs. The external design is conceptually separate from the program code that implements the interface (the internal design). It specifies the interactions between the program and the user or other programs, but not how the program does the required computations. There are typically many different ways of writing a program to implement the same interface.

---

[2] Scott S. Fisher, "Virtual Interface Environments," in Brenda Laurel (cd.), *The Art of Human-Computer Interface Design* (Reading, MA: Addison-Wesley, 1990), p. 423.

[3] Robert M. Carr, "The Point of the Pen," Byte *Magazine,* vol. 16, No. 2, February 1991, p. 211.

[4] Robert J.K. Jacob, "What You Look Is What You Get," *Proceedings of CHI (Conference on Human Factors in Computing Systems), 1990 (New* York, NY: Association for Computing Machinery, 1990), pp. 11-18.

[5] The definition given here reflects the use of the term 'algorithm' in computer science. In applying patent law to inventions involving programmed computers, the courts are required to determine whether the claimed invention is a "mathematical algorithm." The term "mathematical algorithm' was used by the Court of Customs and Patent Appeals to characterize a method of converting binary coded decimal to binary numbers that the U.S. Supreme Court held to be nonstatutory in its 1972 decision *Gottschalk* v. *Benson (409* U.S. 64). The meaning of "mathematical algorithm," and the relationship between 'algorithm' as the term is used in computer science, and "mathematical algorithm," as the term is used in the case law, has been the subject of considerable discussion (see pp. 133-134).

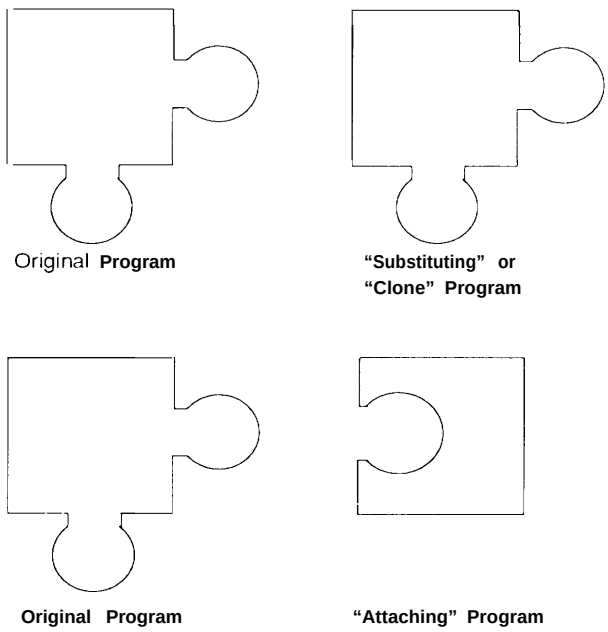
[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (Cambridge, MA: MIT Press, 1990).

The external design will sometimes reflect constraints such as the speed of the processor, the amount of memory available, and the time needed to complete the product. In addition, the process of developing software is iterative—the external design is refined as testing reveals more about user needs or constraints on the implementation.[7]

One example of an external design is the user interface, the conventions for communication between the user and the program. There are also interfaces between programs, such as the ''operating system calls' applications programs use to access functions provided by the systems software of a computer. Communications protocols and the specifications of procedures are other examples of interfaces.

The discussion of appropriate intellectual property protection for interfaces often involves the use of terms such as ''open systems, ' ''interoperability, ' or ' 'compatibility. These terms sometimes have ambiguous meanings.[8] They may be used to describe a situation in which a program from one vendor is able to exchange information with a program from a different vendor. However, these terms are also sometimes used to describe a situation in which multiple vendors offer a product with the same external design. Each of these meanings of ' 'compatibility' implies a different economic effect. For this reason, participants in the software debate sometimes distinguish between ' 'attaching" programs, which are able to exchange information with a program written by a different vendor, and ''substituting programs, which have the same external design (see figure 4-1). ' 'Substituting' programs are sometimes referred to as "workalike" programs or ' 'clones.

Figure 4-l—Comparison of "Substituting" and "Attaching" Programs



SOURCE: OTA.

User Interface

The user interface specifies the interactions between the program and the user. There are a number of different kinds of user interfaces. A programming language is in a sense a user interface—programming using conventional programming g languages is one way to direct a computer to perform a task. For most people this style of interaction is too difficult and inconvenient. By using new technologies, however, different ways of using computers have been developed, Sophisticated, but easy-to-use, user interfaces have created new markets where there are end users who are not programmers.[9] Command languages, menu-based dialogs, graphical user interfaces, and newer interaction techniques have expanded the design choices available to user interface designers.

One interaction style[10] is the *command language* dialog, in which the user issues instructions to a computer through a typed command.[11] The Unix and DOS operating systems usually have this type of user interface. For example, the command for

[7] Daniel S. Bricklin, President, Software Garden, Inc., testimony at hearings before the House Subcommittee on Courts, Intellectual Property, and the Administration of Justice, Nov. 8, 1989, Serial No. 119, p. 221,

[8] ''What Does 'Open Systems' Really Mean?'' *Computerworld*, vol. 25, No. 19, May 13, 1991, p. 21.

[9] Jonathan Grudin, ' 'The Computer Reaches Out, ' *Proceedings of CHI (Conference on Human Factors in Computing Systems), 1990* (New York, NY: Association for Computing Machinery, 1990), pp. 261-267,

[10] For a discussion of different interaction styles, see Ronald M. Baecker and William A.S. Buxton, *Readings in Human-Computer Interaction* (San Mateo, CA: Morgan Kaufmann, 1987), p. 427,

[11] Ibid., p. 428,

Photo credit: Library of Congress

An early user interface. Dr. J. Presper Eckert, Jr. examines the control panel of the ENIAC computer at the University of Pennsylvania in 1946.

deleting a file when a computer is using the Unix operating system is ''rm. ' The difficulty with this type of interface is that it is hard to learn. It may be difficult to remember the available commands, their exact syntax, and how they can be strung together to perform more complex tasks.[12]

A second type of interdiction style uses a *menu.* Instead of having to remember the commands, the user can select choices from a list of alternatives displayed on a screen.[13] The selection can be made by pressing a key that corresponds to a menu option, or by moving the cursor on the screen until the option is highlighted. Menu-based interfaces were made possible by the development of hardware technology that allowed large amounts of information to be quickly displayed on a screen.[14]

Newer interfaces are graphical, using images, in addition to text, to display information to users. *Icons* represent operations (much as menu options do) or data. Commands can be issued by *direct manipulation*[15]*:* instead of using a command language to describe operations on objects, the user ''manipulates' objects visible on the display.[16] The effect of the action is immediately apparent to the user. For example, a user could point to an icon representing a file with the mouse and then ask the system to delete the file; the icon could then disappear from the screen to show the user that the file has been deleted. Graphical user interfaces often allow users to view several activities simultaneously on the screen, through the use of *windows* that subdivide the screen area.

The Design Process

The user interface designer makes many design decisions.[17] Technological change is adding to the range of available choices-color, graphics, sound, video, and animation are only beginning to be explored or widely applied.[18] However, the user interface designer must also work within a set of constraints. Some of these constraints are imposed

12 Bill Curtis, "Engineering Computer 'Look and Feel': User Interface Technology and Human Factors Engineering, " *Jurimetrics,* vol. ***30, No.*** 1, fall 1989, p. 59.

13 Donald A. Norman, ' 'Design Principles for Human-Computer Interfaces, ' *Proceedings of CHI (Conference on Human Factors in Computing Systems), 1983 (New* York, NY: Association for Computing Machinery, 1983), p. 9.

14 John Walker, ''Through the Looking Glass, '' in Brenda Laurel (cd.), *The Art of Human-Computer Interface Design* (Reading, MA: Addison Wesley, 1990).

15 Ben Shneiderman, ' 'Direct Manipulation: A Step Beyond Programming Languages, " *IEEE Computer,* vol. 16, No. 8, August 1983, pp. 57-69.

16 Robert J.K. Jacob, ' 'Direct Manipulation, " in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (New* York, NY: Institute of Electrical and Electronics Engineers, 1986), pp. 384-388.

17 "Unless one clones an existing product, designing even one aspect of an interface—menu navigation, window operations, CO remand names, function key assignments, mouse button syntax, icon design, etc.—gives rise to a potentially endless series of decisions. " Jonathan Grudin, op. cit., footnote 9, p. 261.

18 Ibid.

by the needs of the user, and some are imposed by hardware or software capabilities.[19]

In developing a program, a developer decides which functions the program is to perform for the user. The user interface design reflects the designer's efforts to communicate this functionality to the user. Designing user interfaces is a "communications task like writing or filmmaking."[20] The user interface helps the user develop a ' 'mental model" of how the program works. This mental model does not necessarily reflect the internal 'engineering model' of the program.[21] For example, when a user Points with the mouse to an icon representing a file and asks the computer to delete the file, the user does not have to know where the file is stored or how the hardware and software perform the operation.

One component of the user interface design process is the choice of the interaction style. This choice is influenced by hardware and software constraints, the nature of the application, and the characteristics of the end user. For example, a command language interface may not be appropriate for users who use a program infrequently; a menu interaction style would be easier to use because it would provide the users with reminders of the available commands.

Another component of the design process requires that the functionality of the application be represented within the interaction style. This stage of design would include, for example, the choice of commands and their representation by icons or appropriate mnemonics. This stage of the design reflects the designer's judgment of how the user would want to accomplish certain tasks.[22] For example, the assignment of commands to various menus could reflect a judgment of the relative importance of each command.

Constraints

There are a number of constraints that the user interface designer must take into account. Hardware or software constraints may limit solutions that require too much processing power or are too time-cons uming to program. The capabilities of users present other constraints. Research in the field of human-computer interaction (HCI) tries to find a scientific base for understanding what makes a user interface design successful.[23] HCI research also focuses on methodologies for developing better interfaces.

One of the most important user interface design principles is that ' 'consistency' is important.[24] One type of consistency is internal consistency .25 Internal consistency means, for example, that operations common to several objects in a program have the same results on all of the objects. For example, there might be a single ''delete' command that deletes the selected object, whether it is a text string, a curve, or a file.[26]

From the perspective of intellectual property law, the most significant type of consistency is *external consistency* with the features of other interfaces, i.e., similarity of a given interface to those of other applications and systems. A system is said to be ''backwards compatible'' if it is compatible with an older version of the system, allowing users to benefit

[19] ''[Solutions are shaped by a multitude of problems that arc invisible to those outside of the design process. A wonderfully intuitive solution doesn't matter if the system architecture doesn't support it, or if the resulting code takes up too much memory or runs too slowly. Other problems stem from the basic capabilities of humans, and the requirements of tasks users wish to do. It doesn't matter if the interface responds instantly, if the user can't usc it. Solutions to an interface problem involve compromise. But how do designers determine what an acceptable compromise is? How do designers figure out acceptable tradeoffs between speed and intuitiveness and other seemingly contradictory values and requircmcnts? Thomas D. Erickson, ''Creativity and Design, ' m Brenda Laurel (cd.), *The Art of Human-Computer Interface Design* (Reading, MA: Addison-Wesley, 1990), p. 3.

[20] Paul Heckel, *The Elements {$ Friend/) Software Design* (Alameda, CA: Sybex, 199 1), p, xix. The Epilogue describes the author's experiences in applying for and enforcing a ' 'software patent.''

[21] Donald R. Gentner and Jonathan Grudin, '' Why Good Engineers (Sometimes) Create Bad Interfaces,' *Proceedings of CHI (Conference on Human Factors in Computing Systems), 1990 (New* York, NY: Association for Computing Machinery, 1990), p. 277.

[22] ' '... designers who have only a sketchy or partial understanding of users' tasks will tind it difficult to appreciate the dominant role tasks should play in interface design. in the absence of task analysis, the designer has little to go on and it *bccomcs* convenient to focus on properties of the interface. ." Jonathan Grudin, ' 'The Case Against User Interface Consistency', " *Communications of the ACM, vol.* 32, No IO, October 1989, p. 1165.

[23] Stuart K. Card and Thomas P. Moran, ' 'User Technology: From Pointing to Pondering, ' in Adele M. Goldberg (cd.), *A History of Personal Workstations (New* York, NY: ACM Press. 1988), p. 493.

[24] Ben Shneiderman, *Designing the User Interface* (Reading, MA: Addison-Wesley, 1987).

[25] Grudin, op. cit., footnote 22, p. 1165.

[26] Butler W. Lampson, ''Personal Distributed Computing, '' in Adele M. Goldberg (cd.), *A History of Personal Workstations (New* York, NY: ACM Press, 1 988), p 321

from new features or better performance without having to learn a completely new system. Interoperability and backwards compatibility requirements reflect aspects of the users' experience and environment that should be reflected in an interface design. They are increasingly important as computers are used by more people and in different application areas .27

Program Code

The program code is the implementation of the fiction and external design (including the user interface) of the program. Much effort has been devoted to developing new tools and methods for coping with the complexity of developing new software systems. In general terms, ''software engineering' is concerned with the practical aspects of developing software. It overlaps all of the various specialities of software research, including programming languages, operating systems, algorithms, data structures, databases, and file systems.[28]

The use of high-level programming languages, such as FORTRAN, C, and Pascal, is an important element in the development of complex programs. High-level languages are more powerful than machine languages-each statement in a high-level language usually does the same job as several machine language instructions. In addition, programmers need not be as concerned with the details of the computer's operation, and can write the program in a more natural way. For example, the Pascal-language statement "quantity := total - 5" does the same job as a series of machine language statements; the names of the *variables,* such as "total,' may also suggest something about why the operation is being done. Finally, the programmer does not need to know where the data is stored in memory or how it is represented by patterns of electronic signals.

Programs written in a high-level language must be translated into machine language for execution. This process is called "compilation." The high-level language version of a program is often referred to as ''source code, ' while the machine language version is referred to as "object code. " In the software debate, the relationship between source code and object code has been the subject of considerable discussion (see box 4-A and the "Recompilation" section later in this chapter).

High-level languages also encourage the construction of *procedures, data abstractions,* or *objects,* which allow the decomposition of large and complex programs into smaller pieces that can be attacked independently. Much of the internal structure of a program is due to decisions made by the programmer about how best to decompose the larger problem into smaller pieces. Breaking a larger problem into smaller, more manageable pieces is not unique to software:

> Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.[29]

An example of a procedure might be a sequence of instructions for sorting numbers. This sequence of instructions could be a procedure called "Sort.' "Sort" in effect becomes a new instruction that the programmer can use as if it were an ordinary instruction. At every point in the program where it is necessary to sort some numbers, the programmer simply uses the new instruction, without worrying about the details of its implementation. The programmer only needs to know about the interface that specifies the name of the procedure, its function, and the format in which it exchanges information with other parts of a program.[30]

Software and the Application of Intellectual Property Laws

This section discusses the policy issues associated with the four elements of software outlined in the previous section—program function, external design, the user interface, and the program code. For each of these elements, the section outlines the courts' current approach to its protection, and then discusses arguments for maintaining or modifying the level of protection. To the extent possible, this

---

27 Grudin, op. cit., footnote 22, p. 1171.

28 Computer Science and Technology Bored, National Research Council, *Scaling Up: A Research Agenda for Software Engineering* (Washington, DC: National Academy Press, 1989), p. 18.

29 Harold Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs* (Cambridge, MA: MIT Press, 1985), p. 2.

30 ''Good engineers distinguish between *what* a component does (the abstraction seen by the user) and *how* it does it (the implementation inside the black box). " Jon Bentley, *Programming Pear/s* (Reading, MA: Addison Wesley, 1986), p. 134.

Box 4-A—Authorship

The copyright clause of the U.S. Constitution permits Congress to grant ''authors' exclusive rights to their ''writings.' Before the current Copyright Act and 1980 software amendments became law, there was considerable disagreement as to whether programs were copyrightable "writings." Even after the 1980 amendments were enacted some believed that only high-level language (or ''source code") programs were copyrightable subject matter, while machine language (or 'object code' programs were not protected by copyright law. Some arguments centered on whether code in lower-level languages was *human-readable;* according to one view, only higher-level languages expressed writings (for human readers) eligible for copyright protection.[1]

A corollary debate concerns requirements for ''authorship' of programs. There were two issues: 1) whether "original works of authorship' required a human author, or whether machine-generated works could also be eligible for copyright—the Copyright Office has maintained that the term ''authorship' in the Copyright Act implies a human originator and 2) determining authorship and copyright ownership for machine-generated or machine-assisted works.

Questions of machine authorship have arisen with respect to compilers.[3] When a program is compiled, some information is removed (e.g., comments), some information is added, and the code may then be rearranged to optimize execution speed. From a technological perspective, the end result of the compilation process could therefore be regarded as a ''derivative work' based on the source code program. However, the Copyright Office does not view the object code version as containing sufficient "originality' to be a derivative work, For this reason, the Copyright Office takes the position that the source-code and object code versions of a computer program are copies of the same work.[4]

The final report of the National Commission on New Technological Uses of Copyrighted Works (CONTU) addressed the question of computer-generated works and concluded that ''no special problem exists. The issue continues to be addressed, most recently at a symposium sponsored by the World Intellectual Property Organization (WJPO) on the intellectual property aspects of artificial intelligence. One conferee expressed the view that the real issue was not whether there is a human author, but rather who that author is. While CONTU had concluded that the computer was just a tool to assist a human being in creating a work, the conferee suggested that advances in artificial intelligence meant that the tools were becoming increasingly sophisticated, perhaps indicating a need to apportion authorship among the user and the author of a program used in creating a work.[6] A second participant expressed similar views when discussing the authorship of programs generated by a sophisticated "code generator" from functional specifications.[7]

---

1 The rule that a work must be readable *by a* human *audience* had its origins in *White-Smith Music Publishing CO.v. Apollo Music Co.*, *209* U.S. 1 (1908), which ruled that player-piano rolls could not be copyrighted. For a discussion of the readability requirement, see "Copyright Protection of Computer Program Object Code," Harvard Law *Review*, vol. 96, May 1983, pp. 1723-1744; Christopher M. Mislow, "Computer Microcode: Testing the Limits of Software Copyrightability,' *Boston University Law Review*, vol. 65, July 1985, pp. 733-805; and the dissent of Commissioner John Hersey in National Commission on New Technological Uses of Copyrighted Works (CONTU), *Final Report* (Washington, DC: Library of Congress, July 31, 1978).

2 Cary H. Sherman, Hamish R. Sandison, and Marc D. Guren, *Computer Software Protection Law* (Washington, DC: The Bureau of National Affairs, 1989, 1990), 204.3(d).

3 Ibid., *208.2(b)(5),* and discussion at OTA workshop on "Patent, Copyright and Trade Secret Protection for Software," June 20, 1991.

4 ''The Copyright Office considers source code and object code as two representations of the same computer program. For registration purposes, the claim is in the *computer program* rather than in any particular representation of the program. " Computer Program Practices, *Compendium II, Copyright Office Practices,* Section 321.03.

5 CONTU, op, cit., footnote 1, p. 46.

6 Arthur R. Miller, ''Computers and Authorship: The Copyrightability of Computer-Generated Works, '' in *WIPO Worldwide Symposium on the Intellectual Property Aspects of Artificial Intelligence,* WIPO Publication No. 698 (E) (Geneva, Switzerland: World Intellectual Property Organization 1991) , p. 241.

7 Robert Barr, "Computer-Produced Creations, " in *WIPO Worldwide Symposium on the Intellectual Property Aspects of Artificial Intelligence,* WIPO Publication No. 698 (E), p. 225.

SOURCE: OTA and cited sources.

section separates the question of the appropriate level of protection from the question of how existing patent, copyright, and trade secret laws should be interpreted. In the software debate, policy arguments for a certain level of protection are often characterized as consistent with a ''proper' interpretation of existing law. Legal arguments made in support of policy positions are outlined in the last section of the chapter.

Program Function

Intellectual Property Protection of' Program Function

Existing intellectual property laws are applied to program function in several ways. First, it can be argued that the program function is protected against copying because the implementation is protected by copyright law. Copyright law prevents others from acquiring the functionality of the programmed computer if it is obtained by copying the "expression' in the program code. However, copyright law does not prevent the independent development of a program that performs the same function,

Some aspects of the function of a programmed computer may be protected by maintaining them as trade secrets. For example, a program may be distributed with contractual restrictions on the extent to which it may be studied. Trade secret protection may also be maintained (in part) by distribution of the program in machine language, which is difficult for competitors to understand. This may be a valuable form of intellectual property protection; the scope of protection is the subject of the ' 'decompilation" debate discussed later in this chapter.

Finally, parts of the function of a program may be protected by patent law. The same program may embody many patentable inventions, or none at all, depending on how many parts of the program function are novel, nonobvious, and statutory subject matter. The inventions are claimed either as a series of functional steps carried out by the computer or as a system capable of performing certain functions. The U.S. Patent and Trademark Office (PTO) emphasizes that patents are granted for the functional steps or the system, not the program code .31

When the invention is being claimed as a series of functional steps, or *process, the* applicant does not specify each machine language operation carried out by the computer.[32] Instead, the claim usually describes the steps at a higher level of abstraction, independent of a specific implementation. For example, one patent recites the steps of:

> Identifying a plurality of overlapping working areas on said screen, associating each said working area with an independent computer program, selectively communicating data to each said program. . ,[33]

If, on the other hand, the invention is being claimed as a system or *apparatus, the* applicant describes the 'means' for performing the functions. For example, the apparatus claim corresponding to the process claim described above specifies:

> A computer terminal display system comprising a display surface, means for simultaneously displaying a plurality of overlapping rectangular graphic layers, . . . means for associating each of said graphic layers with an independent computer program. . .

The basis for claiming a software-related invention as an apparatus is that the programmed computer becomes anew machine, or at least a' 'new and useful improvement' of the unprogrammed machine.[34] In an early case that addressed the question of the patentability of software-related inventions, the Court of Customs and Patent Appeals (C. C.P.A.) wrote:

> If a machine is programmed in a certain new and unobvious way, it is physically different from the machine without that program; its memory elements are differently arranged. The fact that these physical

31 ''Recently some commentators have stated that the Office is issuing patents on computer programs or 'Software. ' This is not the case. A 'computer program' is a set of statements or instructions to be used directly or indirectly in a computer to bring about a certain result. A computer program is different from a 'computer process' which is defined as a series of process steps performed by a computer. This distinction may become blurred because some refer to both the series of process steps performed by the computer and the set of statements or instructions as computer programs.'' Jeffrey M. Samuels, Acting Commissioner of Patents and Trademarks, testimony at hearings before the House Subcommittee on Courts, Intellectual Property, and the Administration of Justice, Mar. 7, 1990, Serial No. 119, p. 334.

32 John P. Sumner, "The Copyright/Patent Interface: Patent Protection for the Structure of Program Code, " *Jurimetrics*, vol. 30, No. 1, fall 1989, p. 118.

33 U.S. Patent No. 4,555,775. In order to satisfy the ' 'enablement'' requirement of section 112 of the Patent Act, the specification would show in more detail how the process steps recited in the claim would be performed.

34 *In re Bernhart*, 417 F.2d 1400 (C'. C.P.A. 1969).

changes are invisible to the eye should not tempt us to conclude that the machine has not been changed.[35]

Statutory or Nonstatutory--Novel and nonobvious program function, whether claimed as an apparatus or process, is not necessarily ‘ ‘statutory subject matter for which patents may be granted. In determining whether a claimed computer-related invention is statutory, patent examiners apply the Freeman-Walter-Abele test, outlined in PTO guidelines issued in 1989.[36] This test is named after the appeals court decisions that contributed to developing and refining the test. The purpose of the Freeman-Walter-Abele test is to determine whether a claimed invention is a nonstatutory ‘‘mathematical algorithm’ or mere calculation. [37]

The Freeman-Walter-Abele test applies, in the context of computer-related inventions, the patent law doctrines that regard scientific principles, abstract ideas, and mathematics as nonstatutory. In the 1972 case, *Gottschalk v. Benson,*[38] the U.S, Supreme Court held that a method of converting binary coded decimal (BCD) numbers to binary numbers was not statutory. In C. C.P.A. decisions that followed Benson this method was characterized as a ‘ ‘mathematical algorithm. ’ The Supreme Court objected to the fact that the claimed process just converted numbers from one representation to another without applying the result of the calculation to any other task. Just as a law of nature by itself was not statutory subject matter, the mathematical algorithm was not statutory unless it was applied in some fashion .39

The Freeman-Walter-Abele test has two parts. The first part of the test asks examiners and the courts to determine whether a claim includes a mathematical algorithm. If there is no mathematical algorithm, the claim is for statutory subject matter: ‘‘nonmathematical’ algorithms are statutory.[40] If, on the other hand, a mathematical algorithm is part of the claim, then the examiner must apply the second part of the test and determine whether the algorithm is sufficiently ‘ ‘applied. ’ An invention that includes a mathematical algorithm is statutory only if the mathematical algorithm is ‘ ‘applied in any manner to physical elements or process steps or if the invention is ‘‘otherwise statutory’ without the algorithm.[41]

Mathematical Algorithms-The line between “mathematical algorithms’ and other types of program function is difficult to draw. PTO guidelines state that claims that include mathematical formulae or calculations expressed in mathematical symbols indicate that the program function is a mathematical algorithm. Terms in a claim such as ‘ ‘computing’ or ‘ ‘counting’ may also indicate the presence of a mathematical algorithm.[42] On the other hand, the claim does not recite a mathematical algorithm if the invention can be stated in terms of its operations on things in ‘‘the real world’ that are not conventionally considered ‘ ‘mathematical. For example, claims for inventions that would process architectural symbols[43] or translate languages were found to be ‘ ‘non-mathematical.

The distinction in patent law between mathematical algorithms and other software-related inventions was discussed by the C. C.P.A. in Bradley:

---

35 *In re Bernhart* 417 F.2d 1400 (C.C.P.A. 1969). ‘‘In onc sense, a general-purpose digital computer maybe regarded as but a storeroom of parts and/or electrical components. But once a program has been introduced, the general-purpose digital computer becomes a special-purpose digital computer (i. e., a specific electrical circuit with or without electro-mechanical components) which, along with the process by which it operates, may be patented subject, of course, to the requirements of novelty, utility, and non-obviousness ’ *In re Prater, 415* F.2d 1393, 1403.

36 ‘‘Mathematical Algorithms and Computer Programs,’ *Official Gazette of the Patent Office*, vol. 1106, No. 5, Sept. 5, 1989, pp. 5-12.

37 ‘ ‘The focus of the inquiry should be whether the claim, us *a whole, is* directed essentially to a method of calculation or mathematical formula. ’ *In re Diehr,* 602 F.2d 987,

38 409 U. S. 64.

39 ‘‘It is thus clear that the ‘nutshell’ language of *Benson* expressed the ancient rule that practical application remains key, Because it did not consider the performance of an algorithm by a computer as constituting a practical application o! that algorithm under the rule, the Court must have viewed Benson’s claims as effectively claiming the ‘effect, ’ principle, or law or force of nature (the algorithm) itself. ” *In re de Castelet*, 562 F.2d 1243 (C. C.P.A. 1977).

40 ‘‘The CCPA {has] . . . held that a computer algorithm, as opposed to a mathematical algorithm, is patentable subject matter. ’ *Paine, Webber, Jackson & Curtis v. Merrill Lynch, Pierce, Fenner & Smith, 564* F. Supp. 1358, 1367.

41 ‘‘Ma[hcmatical Algorithms and Computer Programs, ’ op. cit , footnote 36, p. 8.

42 Ibid.

43 *In re Phillips*, 608 F.2d 879 (C. C.P.A. 1979)

44 *In re Toma*, 57 s F.2d 872 (C.C p A.1978).

> [The data being manipulated] may represent the solution of the Pythagorean theorem, or a complex vector equation describing the behavior of a rocket in flight, in which case the computer is performing a mathematical algorithm and solving an equation. This is what was involved in *Benson* and *Flook.* On the other hand, it may be that the data and the manipulations performed thereon by the computer, when viewed on the human level, represent the contents of a page of the Milwaukee telephone directory, or the text of a court opinion retrieved by a computerized law service. Such information is utterly devoid of mathematical significance.[45]

At one point during the period in which it was uncertain how *Benson* was to be interpreted, PTO viewed all claims that involved the use of a computer as ''mathematical' in the *Benson* sense,[46] because computers perform logical and arithmetic operations. The C. C.P.A. responded to this by writing:

> The board's analysis confuses *what the* computer does with *how* it is done. It is of course true that a modem digital computer manipulates data, usually in binary form, by performing mathematical operations. . . But this is only *how the* computer does what it does. Of importance is the significance of the data and their manipulation in the real world, i.e. *what the* computer is doing.[47]

When Is an Invention That Uses a "Mathematical Algorithm " Statutory?—The second part of the Freeman-Walter-Abele test determines if a mathematical algorithm is ''applied' '—in which case the invention is statutory--or if the applicant is attempting to claim the nonstatutory mathematical algorithm. A software-related invention that includes a mathematical algorithm has been found to be applied and statutory if the computer is being used as part of an apparatus or process for transforming physical substances into a different physical state. For example, the invention that the Supreme Court held to be statutory in *Diamond v. Diehr* used the result of a calculation to control a process for curing rubber.

However, a distinction is made between transformations of physical substances and the mere manipulation of ''data. '[48] If the claim is for a series of calculations whose only result is a ' 'pure number,"[49] then the claimed invention is not statutory. For example, the purpose of a process found nonstatutory in *Parker v. Flook*[50] was to calculate an "alarm limit. " Because the alarm limit was just a ''number" and not clearly applied in a physical sense the Supreme Court ruled that the claimed process was not statutory.

At one time it was believed that a mathematical algorithm could become statutory subject matter if the claim were in apparatus form. The apparatus claim was thought to make the invention sufficiently ''applied' or ''non-abstract. ' '[51] However, PTO will no longer approve these applications, viewing them as attempts to circumvent the nonstatutory subject-matter rejection .52

Software Patents—The term "software patent" is frequently used in the policy debate to describe a class of inventions that some believe should not be statutory subject matter or should not be infringing. The policy debate is complicated by the fact that the term software patent' does not correspond directly to a PTO technology class or subclass. The term ''software patent, ' as used in the policy debate, appears to refer to those inventions that would usually be implemented using a program executing on a general-purpose computer.

One difficulty with the use of the term "software patent' is that terminology used by PTO, such as ''computer-related invention' or "computer process' does not refer only to inventions implemented in software. These terms are also used to refer to

---

45 *In re* &a&y, 600 F.2d 812 (C.C.P.A. 1979).

46 See, e.g., *In re Bradley*, 600 F.2d 807 (C. C.P.A. 1979).

47 Ibid., p. 811.

48 "Mathematical Algorithms and Computer Programs, " op. cit., footnote 36, p. 9.

49 *In* r. Waiter, 618 F.2d 767.

50 437 U.S. 584,

51 ''The instant claims, however, are drawn t. physical structure and not to an abstract 'law of nature, a mathematical formula or an algorithm. ' There is nothing abstract about the claimed invention. It comprises physical structure, including storage devices and electrical components uniquely configured to perform specified functions through the physical properties of electrical circuits to achieve controlled results. Appellant's programmed machine is structurally different from a machine without that program. ' *In re Noll*, 545 F.2d 148 (C. C.P.A. 1976).

52 ''Mathematical Algorithms and Computer programs, op. cit., footnote 36, p. 8.

inventions that could be implemented in hardware .53 Under current law, the form of implementation does not determine whether a computer-related invention is statutory-it is significant that the Freeman- Walter-Abele test for statutory subject matter only checks for the use of mathematical algorithms, not the use of software. OTA uses the term ''software-related invention' to refer to inventions implemented in software.

Another difficulty with defining the term "software patent'' is that software is used in a variety of inventions. Not all software-related inventions are products of the ''software industry. ' Many traditional industrial processes are now controlled by computers or use embedded processors. For example, the invention found statutory by the Supreme Court in *Diamond v. Diehr* used a computer to control a process for curing rubber.

''Software patent-type' inventions may be distinguished from other ''computer-related inventions' because they are not as well represented in the PTO'S database of prior art. This database consists mainly of issued patents. Many significant advances in software are not represented in the database because few ''software patents' were issued before the mid-1980s. While in theory these inventions could have been implemented in hardware, in practice they were not. As it was widely assumed that implementation in software precluded the issuance of a patent, few developers applied for patents on these inventions. The gaps in the PTO database of prior art make it more difficult for examiners to determine whether an invention is novel and nonobvious.

When Is ***a Patent for a Software-Related Invention Infringed?***-- There is uncertainty about the scope of protection available from a patent on a software-related invention. The breadth of protection is determined during infringement litigation, of which there has been little to date. One important issue will be the interpretation of claims. Before determining if a device is infringing, courts interpret a claim by looking at the specification and the prosecution history. For exampIe, "means-plus-function' claims do not cover all of the "means" for performing a function, only the "structure, material or acts described in the specification and equivalents thereof. ' '[54] One question might be whether the claims cover both hardware and software implementations of an invention.

"Literal infringement' occurs when an accused device or process incorporates each and every element of a claim. Even if there is no literal infringement, a device can still be found to be infringing-this is known as the "doctrine of equivalents, '[55] The doctrine of equivalents is applied when an accused device does not incorporate every element of the claim but is still ''close enough.'[56] Infringement occurs if the accused device or process accomplishes "substantially the same thing, in substantially the same way to achieve substantially the same result. ' '[57] Generally, ' 'pioneer' inventions that represent a substantial advance over the prior art are granted a broader range of equivalents by the courts. Observers have argued that some software-based patents, though claimed and allowed broadly due to a lack of knowledge of constraining prior art, may not be true pioneers in their fields .58

Protection of Program Function— "Software Patent" Policy Issues

There has been considerable debate about the granting of patents for software-related inventions. Some believe that no inventions that use software should be patentable or that only software-related inventions that are traditional industrial processes should be statutory subject matter.[59] Others believe that inventions that use software should not be

53 Whether Or not a software implementation infringes a "hardware" patent depends on the interpretation of "equivalent" in the [35 U. S. C.] section 112(6) sense and the "doctrine of equivalents. " See Ronald S. Laurie and Jorge Contreras, '*Application of the Doctrine of Equivalents to Software-Based Patents, '' in Michael S. Keplinger and Ronald S. Laurie (eds.), *Patent Protection for Computer Software* (Englewood Cliffs, NJ: Prentice Hall Law and Business, 1989), p. 161.

54 35 U.S.C. 112(6).

55 The ''doctrine of equivalents' should not be confused with the meaning of ''equivalent as used in interpreting means-p s-function claims. See Donald S. Chisum, *Patents* (New York, NY: Matthew Bender, 1991), vol. 4, pp. 18-60—18-63.

56 Laurie and Contreras, Op, cit., footnote 53, p. 163.

57 Chisum, op. cit., footnote 55.

58 Laurie and Contreras, op. **cit.,** footnote 53, p. 169.

59 See Pamela Samuelson, "Benson Revisited, " *Emory Law Journal, vol. 39, No. 4,* fall 1990, pp. 1133-1142.

treated differently from other types of inventions.[60] In fact, some who believe that software-related inventions should be patentable have argued that many of what are now deemed nonstatutory mathematical algorithms should be statutory subject matter.[61]

Some of the concerns about the patenting of inventions that use software are similar to those expressed about the patent system in general[62] or about the patent system's ability to accommodate any new technology. In some cases these questions have been brought into sharper focus in the context of software. For example, the appropriate length of the patent term has been a subject of discussion for many years (see chapter 6); many believe that 17 years is especially inappropriate for a fast-moving technology such as software.[63] The fact that patent applications are kept secret until the patent issues makes it impossible to be certain that a product under development will not 'be accused of infringing a patent; this ''kmdmine' problem may be exacerbated by the longer pendency for computer-related inventions. However, two areas of concern are more directly related to the question of software patentability: the effect of patents on industry structure and innovation, and the quality of the patents that have been granted.

First, it is argued that the widespread use of patents could change the structure of the software industry in a way that would actually reduce the rate of innovation. According to those who hold this view, patenting favors larger companies, not the small companies that have historically been the source of much innovation. The software industry has had a disproportionate number of smaller companies; in part, this industry structure was due to the fact that the limited use of patents and licenses kept barriers to entry modest. There is a concern that the widespread use of patents could reduce small-company-based innovation by raising barriers to entry, either as a result of the need to pay royalties or the added costs of searching and filing for patents,[64] In addition, *large companies could engage in* portfolio trading while small companies without extensive patent portfolios would have their freedom to develop products restricted.[65]

The alternative view is that the economics of the software industry is not that different from the economics of other industries, and that patents are therefore equally appropriate for encouraging software development. Some argue that software development is becoming increasingly expensive, and patents may provide the incentive needed to invest or attract venture capital funding.[66] In addition, it may be that patents in fact benefit small companies, by providing a means to protect their development effort against appropriation by a larger company.[67]

A second set of concerns focuses on the quality of patents that have issued. Some consider that many patents have issued that do not in fact represent significant advances.[68] From the developer's perspective, this increases the probability that a program could be accused of infringing patents. The developer would then have to decide whether to engage in costly litigation in an attempt to invalidate the patent. The perceived problems with examination quality have primarily been the result of the long period of time during which it was uncertain whether software-related inventions were statutory subject matter. Few patents issued for software-related inventions, leading to gaps in PTO'S database of prior art. Some believe that the the problems with the database of prior art can be resolved given enough time.[69] However, the burdens on the PTO of increasingly backlogged applications and external criticism may be exacerbated over the next several

60 Donald S. Chisum, "The Patentability of Algorithms, ' *University of Pittsburgh Law Review,* vol. 47, No. 4, summer 1986, pp. 1009-1019.

61 Ronald S. Laurie, ''The Patentability of Artificial Intelligence Under US Law, '' in Morgan Chu and Ronald S. Laurie (eds.), *Patent Protection for Computer Software* (Englewood Cliffs, NJ: Prentice Hall Law and Business, 1991), pp. 288-290.

62 For an overview of current concerns about the patent system, see ' 'The Patent Game: Raising the "rit e* " *Science, vol.* 253, July 5, 1991, pp. 20-24.

63 Mitchell D. Kapor, Chairman and Chief Executive Officer, ON Technology, Inc., testimony at hearings before the House Subcommittee On Courts, Intellectual Property, and the Administration of Justice, Nov. 8, 1989, Serial No. 119, p. 244.

64 Brian Kahin, ' 'The Software Patent Crisis, ' *Technology Review,* vol. 93, No. 3, April 1990, p. 53.

65 Ibid.

66 Chisum argues that software should not be a ''disfavored technology. " Donald S. Chisum, op. cit., footnote 60,

67 Elon Gasper, Ed Harris, Paul Heckel, William Hulbig, Larry Lightman, and Mike O'Malley, letter, *New York Times,* June 8, 1989, editorial page.

68 Kahin, op. cit., footnote 64.

69 David Bender, letter, New *York Times,* June 8, 1989, editorial page.

Box 4-B--Cryptography

The recorded history of cryptography as a means for securing and keeping private the content of communications is more that 4,000 years old. Manual encryption methods using code books, letter/number substitutions and transpositions, etc. have been used for hundreds of years-the Library of Congress has letters from Thomas Jefferson to James Madison containing encrypted passages. Modem (computer-based) cryptographic technologies began to develop in the World War II era, with the German *Enigma* cipher machine and the successful efforts to break the cipher computationally; cryptographic research and development in the United States has often proceeded under the aegis (or watchful eye) of the National Security Agency and, to some extent, the National Institute of Standards and Technology (NIST).

Encryption techniques can be used to maintain the secrecy/privacy of messages; they can also be used to authenticate the content and origin of messages. The latter function is of widespread commercial interest as a means of electronically authenticating and ' 'signing" commercial transactions like purchase orders and funds transfers, and ensuring that transmission errors or unauthorized modifications are detected. Encryption is a mathematical process and the descriptions of different techniques (e.g., the Federal Data Encryption Standard (DES), the Rivest-Shamir-Adelman (RSA) public-key cipher, the "Trapdoor Knapsack" cipher, etc.) are usually referred to as ''algorithms. ' Nevertheless, cryptographic systems have been successfully patented (usually as "means plus function" claims); the RSA system was patented by the Massachusetts Institute of Technology and licensed in 1982 to the inventors, who formed a private company to market the system. Results from other university research in cryptography have also been patented and licensed—for example, U.S. Patent No. 4,218,582 for a "Public Key Cryptographic Apparatus and Method" invented by Martin Hellman and Ralph Merkle of Stanford was granted in 1980----as have commercially developed systems.

Patents may be complicating development of a new Federal standard for a public-key cipher. In 1991, NIST proposed a digital signature standard (DSS) for unclassified use in digitally authenticating the integrity of data and the identity of the sender of the data. The proposed standard is intended to be suitable for use by corporations, as well as civilian agencies of the government. NIST has filed for a U.S. patent on the selected technique and plans to seek foreign patents. NIST has also announced its intention to make the DSS technique available worldwide on a royalty-free basis. According to press accounts, NIST has chosen the DSS algorithm as a standard to avoid royalties. Some critics of this choice (including the company marketing the RSA system) have asserted that the RSA algorithm is technologically superior and that NIST deliberately chose a weaker cipher. In late 1991, NIST'S Computer Security and Privacy Advisory Board went on record as opposing adoption of the proposed DSS.

SOURCES: U.S. Congress, Office of Technology Assessment, *Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information,* OTA-CIT-31O (Washington, DC: U.S. Government printing Office, October 1987); Michael Alexander, "Data Security Plan Bashed,' *Computerworld,* vol. 25, No. 26, July 1, 1991, pp. 1, 80; Richard A. Danca "NIST Crypto Change Takes Fed Vendors by Surprise," *Federal Computer Week,* July 8, 1991, pp. 1,37; *Federal Register,* vol. 56, No. 169, Aug. 30, 1991, pp. 4298042982; Richard A. Danca "NIST Signature Standard Whips Up Storm of Controversy From Industry," *Federal Computer Week,* Sept. 2, 1991, p. 2; Darryl K. Taft "Board Finds NIST'S DSS Unacceptable, " *Government Computer News,* vol. 10, No. 26, Dec. 23, 1991, pp. 1,56.

years. Computer implemented processes will become more "commonplace and important in a wide variety of industries and applications (see box 4-B), ranging from home entertainment to scientific research to financial services.

There may be practical limitations on attempts to exclude ''software inventions from the patent system. First, many claims in computer-related invention patents issuing today cover both hardware and software implementations; if the software implementation were not an infringement, the value of a "hardware" invention could be appropriated. Some inventions that were initially "hardware" inventions are now being implemented in software, as faster processors have become available. Second, there are-many inventions that use software but are not the type of invention that has been the subject of concern in the policy debate. There does not appear to be much concern about the patenting of traditional industrial processes that happen to use software as part of the apparatus or to perform a step in the process.

The ***Freeman-Walter-Abele*** *Test—Another issue* is whether the Freeman-Walter-Abele test draws the line between statutory and nonstatutory subject matter in the right place. Some observers believe that some of what PTO and the Court of Appeals for the Federal Circuit regard as nonstatutory mathematical

algorithms should be statutory subject matter.[70] A variant of this opinion is that "field of use" limitations should be sufficient to demonstrate that a mathematical algorithm is ''applied' and the claimed invention statutory. In *Parker v. Flook*[71] *the* Supreme Court held that language in the claim that limited the field of use of a mathematical algorithm to processes comprising the catalytic conversion of hydrocarbons was not sufficient to make the invention statutory.

Supporters of an expansion in the scope of statutory subject matter argue that some mathematical algorithms fall within the ''useful arts' and that their invention should be encouraged by the patent system. Under the *Benson* analysis, more efficient methods of solving general ' 'mathematical' problems on a computer are not statutory subject matter. In its decision finding Benson's application to be statutory (later reversed by the Supreme Court), the C. C.P.A. listed a number of advances embodied by the invention:

> . . . reducing the number of steps required to be taken, dispensing with the repetitive storing and retrieval of partially converted information, eliminating the need for interchanging signals among various equipment components, and the need for auxiliary equipment, and decreasing the chance of error.[72]

The Freeman-Walter-Abele test may also be difficult to apply consistently .73 The distinction between "mathematical" and "nonmathematical" algorithms has been criticized by computer scientists as a creation of the case law that lacks a foundation in computer science.[74] It may be that:

> . . . any attempt to find a helpful or cutting distinction between mathematics and nonmathematics, as between numerical or nonnumerical, is doomed.[75]

Some commentators have suggested that patents have issued for inventions that do not appear to satisfy the conditions of the test, or at least indicate that the test is difficult to understand.[76] There is also a sense among some patent attorneys that PTO has recently changed its application of the Freeman-Walter-Abele test, resulting in an increase in the number of rejections for nonstatutory subject matter.[77]

The difficulty in distinguishing between mathematical and other algorithms has been used to support calls for both an expansion and a reduction in the scope of statutory subject matter. Those who would reduce the scope of statutory subject matter argue that, since the distinction cannot be easily made, all algorithms should be nonstatutory .78 Those who would expand the scope argue the opposite—if some algorithms are statutory and no distinction can be made, then statutory subject matter should include many of what are now called mathematical algorithms.[79]

External Design

When Is the External Design of a Program Protected?

The external design of a program includes its user interface and the conventions for communication with other programs. The design of a user interface can include the appearance of images on a screen, the choice of commands for a command language, or the design of a programming language. The external design may also include file formats and communi-

70 ''[P]olicy considerations indicate that patent protection is as appropriate for mathematical algorithms that are useful in computer programming as for other technological innovations. ' Chisum, op cit., footnote 60, p. 1020.

71 98 S. Ct. *2522 (1978).*

72 *In re Benson,* 441 F.2d 682,683 (C. C.P.A. 1971).

73 "Maintenance of such an arbitrary and unclear line between mathematical and nonmathematical algorithms is necessary only because of the assumption of the continued vitality of *Benson. Benson* held that 'something' is per se unpatentable but failed to provide reasoning that could bc applied to determine the scope of the per se rule. ' Chisum, op. cit., footnote 60, p. 1007.

74 Allen Newell, ' 'The Models Are Broken, The Models Are Broken, ' *University of Pittsburgh Law Review, vol. 47,* No. 4, summer 1986, pp. 1023-1035,

75 Ibid., p. 1024.

76 on, issued patent often cited involves the ''Karrnakar algorithm. ' See Pamela Samuelson, ''Benson Revisited, '' *Emory Law Journal, vol.* 39, No. 4, fall 1990, pp. 1099-1102.

77 Robert Greene Sterne and Edward J. Kessler, ''Worldwldc Patent Protection in the 1990's for Compulcr-Related Technology, ' in Morgan Chu and Ronald S. Laurie (eds.), *Patent Protection for Computer Software* (Englewood Cliffs, NJ: Prentice Hall Law and Business, 1991), p. 445.

78 See, e.g., Samuelson, op. cit., footnote 76, pp. 1139-1 140.

79 See, e.g., Chisum, op. cit., footnote 60, p. 959.

cations protocols. Modules of a program, such as a procedure, also have an interface.

Patent, copyright, and trade secret law have all been used to protect elements of external designs. Some external designs, such as communications protocols, may be patentable.[80] Patents can also be used to protect elements of user interfaces, if novel and nonobvious, and design patents may be available for some of the ''ornamental' aspects of user interfaces.[81] Trade secret law may provide some degree of protection, if a program is distributed in machine language form. To specify all of the externally observable behavior of an interface, one must generally know all permitted sequences of interface actions. Determining all of the possible sequences of interface actions may be difficult if it is not possible to study the assembly language or high-level language versions of the program code.

Copyright protection may be available for aspects of external designs, especially those that use screen displays. The screen displays of a video game are often protected through an ' 'audiovisual' copyright. Other user interfaces have also been found to be protected by copyright law, There are two different approaches to protecting user interfaces using copyright law. One approach is to protect the user interface through the copyright in the underlying program.[82] The other approach is to regard tie screen display as a separate work from the program code, and protect it as an audiovisual work or as a compilation of literary terms (for interfaces that use text). The scope of copyright protection for user interfaces that do not use a screen display, such as command languages or programming languages, has not been at issue in a decided case. However, there are some who feel that the legal reasoning used in the cases where the user interface used screen displays would protect these types of external design as well.[83]

Interfaces other than user interfaces have also been the subject of copyright litigation. The format for entering statistical data into a structural-analysis program has been found to be not protected.[84] There has been an effort to assert copyright protection for what the court described as ''minor content variations' in the bit pattern of a communications protocol,[85] *but the court did* not find in the variations ''choice and selection' beyond the content of an earlier protocol to evidence sufficient originality. Dictum in a 1985 case, *E.F. Johnson v. Uniden,*[86] indicates that the court viewed the development of a communications product (a radio) ''compatible' with an existing product as permissible behavior. The court emphasized, however, that permissible development of a compatible product requires that the implementation be done independently[87]—achieving compatibility in the external design is not an excuse for copying the program code.[88] No cases have addressed such issues as the interfaces in class libraries for object-oriented languages.

Perceptions of the scope of copyright protection for interfaces have changed over the past decade. In the early 1980s some had assumed that the external design was unprotected[89] and that the only issue was whether the implementation had been done independently—there are usually different ways of writing a program with the same interface. The view that copyright protection for interfaces was limited was reflected in the use of 'clean rooms, ' in which a specification of the program is given to program-

80 Sterne and Kessler, op. cit., footnote 77, p. 402.

81 Daniel J. Kluth and Steven M. Lundberg, ''Dcslgn Patents: A New Form of Intellectual Property Protection for Computer Software, ' *JPTOS*, December 1988, p. 847.

82 *Telemarketing* v. *Symantec*, 12 U, S.P.Q.2d1991 (N.D. Cal. *1989); Lotus v. Paperback*, 740 F.Supp. 37 (D. Mass. 1990).

83 Ronald L. Johnston and Allen R. Grogan, ' 'Copyright Protection for Command Driven Interfaces, ' *The Computer Lawyer*, vol. 8, No. 6, June 1991, p. 1.

84 *Engineering Dynamics* v. *Structural Software*, No. CV 89-1655, U.S. District Court, E.D. Louisiana, Aug. 29, 1991; *Synercom Technology* v. *University Computing*, 462 F. Supp. 1003 (N.D. Tcx. 1978).

85 *Secure Services Technology* v. *Time and Space Processing*, 722 F. Supp. 1354, 1362 (E.D. Va. 1989).

86 623 F.Supp. 1485 (D. Minn. 1985).

87 Ibid,, p. 1501, footnote 17.

88 S.. also *Apple v. Franklin*, 714 F.2d 1240, 1253 (3d Cir. 1983).

89 In a 1986 article Duncan Davidson wrote: ''It is striking that despite all the concerns raised over software copyrights, a patent-like monopoly does not exist m any area of software. Application environments like Lotus 123 have been both cloned and emulated by other spreadsheets. " Duncan M. Davidson, ''Common Law, Uncommon Software, *University of Pittsburgh Law Review*, *vol. 47, No. 4,* summer 1986, p. 1077. (In the 1990 case *Lotus* v, *Paperback* (740 F. Supp. 37) such a clone was found to be a copyright infringement.)

mers who have not seen the original program (the aim of the procedure being to make available ''clean' uncopyrighted ideas without the ''taint' of the program's copyrighted expression). The theory was that since the clean-room programmers had not seen the original program there could be no infringement. The most commonly cited example of a clean-room developed product is an operating system program used in a type of microcomputer.[90] The legal status of clean-room practices is still uncertain; however, in the 1991 case *Computer Associates v. Altai,* a program implemented using a clean-room-type process was found to be noninfringing.

Policy Issues-External Design

The economic effects of protecting interfaces are difficult to evaluate, requiring a determination of the appropriate level of incentives and the role of standards and network externalities. An evaluation of the economic effects of intellectual property protection may also be complicated by the fact that there are different types of interfaces. The value of a standard, and the balance between the cost of designing the interface and cost of its implementation, may both depend on the type of interface.

lncentives-One policy position is that intellectual property protection is required in order to provide the proper incentives for the development of software. It is argued that protection of the program code alone is not sufficient to provide this incentive. Because there are different ways of writing a program with the same interface, it may be possible to reimplement the same interface without a finding of infringement, If the cost of reimplementation were small when compared to the orginal developer's investment in designing the interface, it would be relatively easy to appropriate this investment. Without more direct intellectual property protection for the external design, it is argued, there would be less incentive to develop new interfaces.

An important factor in evaluating whether external designs should be protected is therefore the relative cost of design and implementation. Supporters of intellectual property protection for external designs argue that the cost of implementation is becoming less significant. In *Lotus the* court said:

> I credit the testimony of expert witnesses that the bulk of the creative work is in the conceptualization of a computer program and its user interface, rather than in its encoding.[91]

Similar considerations are said to apply to other types of interfaces: during an intellectual property panel at the 1990 Personal Computing Forum, one participant said ' 'the hard work in doing object-oriented technology is in the interface design, The implementation of an object is trite. ' '[92] The relative cost of design and implementation is also an important factor in the recompilation debate discussed later in this chapter—it has been argued that recompilation can make it significantly less expensive for a competitor to reimplement an existing program.

The alternative view is that there is sufficient incentive to engage in the design of interfaces, even without intellectual property protection. Those who argue for this position claim that reimplementation may be time consuming and expensive, providing the original developer with significant lead time.[93] Other factors may also provide a significant advantage to the original designer of the interface.[94] Long-range planning of enhancements may favor the interface originator, for example.[95]

---

90 Ibid.

91 *Lotus* v. *Paperback,* 740 F. Supp. 37,56 (D. Mass. 1990). Others also hold the view that the effort involved in dcsigrnng a user interface deserves protection. See, e.g., Ben Shneiderman, "Protecting Rights in User Interface Designs, ' *SIGCHI Bulletin,* October 1990, vol. 22, No. 2, p.18.

92 Adele Goldberg ParcPlace Systems, at Personal Computing Forum 1990, transcript in *Release 1.0,* vol. 90, p. 107.

93 ''Software is so complex and idiosyncratic that, unless the person is deliberately copying the internals of the code, a reproduction of a sophisticated application so flawless that it has equivalent quality and utility to the original is usually significantly difficult and expensive to produce that any firm with the economic and intellectual resources to do a good job at this prefers to create original products which represent a greater opportunity. ' Mitchell D. Kapor, Chairman and Chief Executive Officer, On Technology, Inc., testimony at hearings before the House Subcommittee on Courts, Intellectual Property and the Administration of Justice, Nov. 8, *1989,* Serial No. 119, p. 243.

94 ' 'And I must tell you that the de velopment of the software program, maintaining it, keeping it documented, porting it to other computers, evolving it, enhancing it, supporting it, answering service calls, and so on and so forth, this is the bigger picture, and it is really simplistic (o say that if one can actually just lake the looks of a program, they will be able to run with it and in fact surpass whoever originated the first program. " Richard Bezjian, President, Mosaic Software, at panel on ' 'Intellectual Property in Computing: (How) Should Software Be Protected" (Cambridge MA: Transcript, Oct. 30, 1990), p. 24.

95 Brett L. Reed, ''Obscrvatioms on the Economics of Copyright and User Interfaces, " *International Computer Law Adviser,* vol. 5, No. 10, July 1991, p. 4.

Network Externalities-There is a question as to whether the effect of intellectual property laws on standardization should affect an evaluation of the appropriate level of protection for interfaces,[96] Standards benefit users in a number of ways. For example, a greater variety of application programs will be developed if there is a standard operating system-developers will be able to sell to a larger market and more easily recover their development costs. Another example of an advantage of standards is that consistency among user interfaces makes it easier for users to learn to use a new program. The benefits to users that result from the wider use of an interface are known as ' 'network externalities' (see also ch. 6). Moreover, users may benefit from competition among suppliers of standard product. For example, suppliers of compilers for standard languages compete on the basis of the cost of the compiler and the efficiency of the machine language code generated.

De facto standards evolve through the actions of the market. If there is a dominant firm, the interface that it has developed is more likely to become the standard. Alternatively, a de facto standard can develop because of a ' 'bandwagon' effect. If consumers are faced with a choice between different interfaces, network externalities make the more widely used product more attractive. Consumers value the network externalities, not just the intrinsic value of the interface.

Standards may also be negotiated using standards committees. Firms engage in voluntary standards-setting when they determine that they are better off with a part of a larger market than if they were to continue trying to establish their interface as a de facto standard. Consumers may be less willing to buy a proprietary product. For example, it is not clear whether a computer language available from a single vendor would be widely used[97]—a developer might be unwilling to rely on a single supplier.

One view is that intellectual property protection may harm users by affecting standardization processes. It is argued that firms may not have the correct incentives to engage in voluntary standards setting because intellectual property protection can increase a fro's vested interest in seeing the interface it has developed chosen as a standard, slowing the standardization process.[98] This could harm users, until a standard is negotiated or one interface prevails in the marketplace, Users could also be harmed if new programs are not "backwards compatible" and require users to learn a new interface to take advantage of new features or better performance, In addition, it has been argued that network externality effects can complicate the balancing of incentives for software development, by resulting in "extra" revenues for firms that succeed in establishing their products as a de facto standard and making it more difficult for other firms to enter the market.[99]

The other view is that the question of standards should be kept separate from the basic issue of the proper incentives for software development. Furthermore, it is argued that voluntary standards efforts are sufficient,[100] and that there is a trend in the computer industry toward using more formal standardization and licensing processes. Consortia have formed in a number of areas, such as user interface design and operating systems. There are a variety of voluntary standards committees that are developing standards for data communications protocols,[101] operating system interfaces,[102] and principles for user interface design.[103]

---

[96] For discussions of standardization considerations, see Peter S. Menell, ''An Analysis of the Scope of Copyright protection for Application Programs, ' *Stanford Law Review, vol. 41, No. 5, May 1989,* pp. 1100-1 101; Richard H. Stem, ''Legal protection of Screen Display s,' *Columbia 'LA Journal of Law& the Arts,* vol. 14, pp. 291-292; Anthony L. Clapes, *Software, Copyright, & Competition* (Westport, CT: Quorum Books, 1989), p. 206.

[97] Alfred Z. Spector, "Software, Interface, and Implementation, " *Jurimetrics*, vol. *30, No.* 1, fall 1989, p. 89.

[98] Joseph Farrell, "Standardization and Intellectual Property, " *Jurimetrics*, *vol. 30, No.* 1, fall 1989, p. 44.

[99] The court in *Lotus* did not view this as affecting the determination of whether [he copyright had been infringed,

100, 'Th, exclusions [of ,nte-faces and limitations of 'decompilation' from copyright law] are unnecessary to permit development of 'interoperable' programs: thousands of such programs have been created under the existing copyright rules, thanks to the work of international standards organization and the voluntary sharing of necessary information. " William T. Lake, John H. Harwood, and Thomas P. Olson, "Tampering With Fundamentals: A Critique of Proposed Changes in EC Software Protection, ' The *Computer Lawyer*, vol. 6, No. 12, December 1989, p. 3.

[101] Steven Turner, "The Network Manager's Compendium of Standards, " *NetworkWorld*, vol. 8, No. 15, Apr. 15, 1991, p. 1.

[102] D. Richard Kuhn, ''IEEE's Posix: Making Progress, ' *IEEE Spectrum, vol.* 28, No, 12, December 1991, pp. 36-39.

[103] See, e.g., Pat BillingsIcy, ' 'The Standards Factor: Standards on the Horizon, " *SIGCHI Bulletin,* vol. 22, No. 2, p.10.

User Interface

Which Elements of a User Interface Design Have Been Protected?

The type of interface that has received the most attention in the software intellectual property debate is the user interface. Two factors have been taken into account by the courts when determining the scope of protection for user interfaces. First, standard ''building blocks' of user interface design such as the idea of using a menu have not been protected. Second, the courts have recognized constraints on the design; for example, commands that are necessary to the overall purpose of a program have not been protected. Hardware and software constraints on the way information is presented on the screen have also been recognized,

Having determined which elements of an interface are either standard building blocks or imposed by technical constraints, the courts look for design choices. When there are design choices remaining after the constraints have been taken into account, the courts generally protect the elaboration of these design choices into a user interface. The choice and organization of commands in a menu hierarchy, and the arrangement of command terms on a screen, for example, have been found to be protected.

Unprotected Elements.—*In* general, common interaction techniques have not been protected. The idea of using a menu has not been protected. Particular menu styles have also not been protected by copyright, on the grounds that they were common in the industry. The use of a pull-down menu was not protected expression in *Telemarketing.*[104] *The* use of a two-line moving cursor menu was described as "functional and obvious" (and not protected) in Lotus.[105] Also found unprotected have been standard ways of entering commands,[106] selecting menu entries,[107] and navigating on the screen.[108]

Commands and menu options required for the overall purpose of the program would probably not be protected. For example, in *Telemarketing* menu options that allowed the user to access existing files, edit work, and print the work were not protected.[109] Also not protected were functions that were likely to be found in any outlining program, or cost-estimation program.[110] The rimes chosen for individual menu entries have, in general, not been protected. For example, the use of "print" as the command name for printing would probably not be protected.111

The courts have also addressed the issue of the organization of information on the screen, and have generally recognized constraints. Centering the headings on a screen, locating program commands at the bottom of the screen, and the use of a columnar format have all been found to be either unprotected ''conventions' chosen from a narrow range of choices or not original.112

Protected Elements.—What *has* generally been protected is the overall set of command terms and their organization into menus. The designer's judgment of the way in which users would want to use a spreadsheet, as reflected in the ' 'menu structure, " including the overall structure, the order of commands in each menu line, and the choice of letters, words, or ''symbolic tokens' to represent each

---

104 ''Plaintiffs may not claim copyright protection of an idea and expression that is, if not standard, then commonplace in the computer software industry. " *Telemarketing v. Symantec, 12* U. S. P.Q.2d 1991, 1995 (N.D.Cal. 1989).

10574.0 F. Supp. 37, 65 (D. Mass. *1990).*

106 "...*the* typing of two symbols to activate a specific command is an 'idea. ' " *Digital v. Softklone, 659* F. Supp. 449,459 (N.D. Ga. 1987).

107 *MTI* v. *CAMS,* 706 F. Supp. 934, 9950. Conn. 1989).

108 '' [T]he idea at issue, the process or manner of navigating internally on any specific screen displays likewise is limited in the number of ways it may be simply achieved to facilitate user comfort. To give the plaintiff copyright protection for this aspect of its screen displays, would come dangerously close to allowing it to monopolize as ignificant portion of the easy-to-use internal navigation conventions for computers. *MTI* v. *CAMS,* 706 F. Supp. 984, 995 (D. Conn. 1989).

10912 U. S, P.Q.2d 1991, 1995 (N.D.Cal.1989).

110 ''Nor is the listing of items for which &U is supplied subject to copyright, because, in the language of the machining industry, speeds and feeds, machining times and costs, and data specific to the size, depth, and diameter of the hole is all closely related to and hence incident to the idea of displaying this data. . ." *MTI* v. CAMS, 706 F. Supp. 984, 998 (D. Corm 1989).

111 ' 'Obvious' command terms which merge with the idea of the command term were discussed by the court in *Lotus. Lotus v. Paperback, 740* F. Supp. 37,67 (D. Mass. 1990).

112 *MTI* v. *CAMS,* 706 F. Supp. 984, 994-5 and 998 (D. Conn. 1989).

command was found to be protected in *Lotus v. Paperback.*[113] *In MTI v. CAMS, the* designer's view of how a user would go through the process of cost-estimating, as reflected in the sequence of menu screens, was found to be protected.[114] The existence of design choices has frequently been shown by the existence of a third program that uses a different menu structure and has different commands.[115] This has been interpreted to show the absence of ' 'mechanical or utilitarian constraints' [116] on the designer.

In one case, the commands themselves were not protected, but the arrangement of the command terms on the screen was protected expression.[117] As a result, the defendant was forced to redesign the product to present the command options to the user in a different way. Instead of presenting the command options on a single screen, they were distributed over a sequence of screens.

Standards—Industry conventions such as the use of certain menu styles, or the use of the 'return' key to select a highlighted menu element have not been protected by copyright law. On the other hand, the choices made by the designer of a successful product in developing the menu structure have not been recognized as a constraint on later developers. In *Lotus* the defendants sought to show that while there may have once been a number of design choices, the success of the plaintiff's spreadsheet product in the market sharply limited the choices of later developers, due to network externality effects. This argument was not accepted; the court wrote:

> By arguing that 1-2-3 was so innovative that it occupied the field and set a de *facto* industry standard, and that, therefore, defendants were free to copy plaintiff's expression, defendants have flipped copyright on its head. [118]

User Interface—Issues

At one level the software intellectual property debate has been concerned with the question of whether user interfaces should be protected at all. The secondary issues have focused on the question of which elements should be protected. Intellectual property law establishes rules for competition in user interface design by drawing lines between protected and unprotected elements. The debate about ''look and feel' '[119] reflects a concern that a particular *style* of interface would be protected by copyright law. There is a concern that the protection of an interaction style would leave too little room for innovation by others within the general style, or for its use in a different program.

However, in cases decided so far, the courts have held the mere use of a menu-based interaction style to be unprotected. The use of the "spreadsheet metaphor' has also been held to be unprotected. In effect, the courts have viewed the use of these common types of interaction in the same way that they view the use of words and stock characters alone in the application of copyright law to literature: as building blocks that should not be protected. To give one creator a monopoly over these basic elements would effectively stunt the efforts of other creators to elaborate on these elements in the production of their own works.[120]

One difficulty is that technological change is continually adding new building blocks. The cases that have been decided all involved simple text-based menus that do not represent the state of the art in user interface design. Some of the cases now in the courts involve graphical user interfaces, and it is less clear what constitutes an unprotected ''building block" of graphical user interface design, and what constitutes an elaboration of building blocks into a

113740 F. Supp. 37,67 (D. Mass, 1990). The court said that '' [a]n example of distinctive details of expression is the precise 'structure, sequence, and organization'. .of the menu command system. '

114 706 F. Supp. 984, 994 (D. Corm. 1989).

115 ''In the present case, the Court has already noted that the existence of 'Stickybear Printer' [a third program] disproves defendant's argument that there are a very limited number of ways to express the idea underlying 'Print Shop. ' Thus, there is no danger in the present case that affording copyright protection to the 'instructions' of 'Print Shop' will amount to awarding plaintiff a monopoly over the idea of a menu-driven program that prints greeting cards, banners, signs and posters. " *Broderbund Software* v. *UnisonWorld, 648* F. Supp. 1127, 1134 (N.D. Cal. 1986).

116 *Broderbund Software v. UnisonWorld, 648* F. Supp. 1127, 1133 (N.D. M. 1986).

117 *Digital Communications Associates v. Softklone Distributing, 659 F.* Supp. 449 (N.D. Ga. 1987).

118 740 F. Supp. 37, 79 (D. Mass. 1990).

119 Despite its 'widespread use in public discourse, a court has said that the " 'look and feel' concept, standing alone [was not] significantly helpful" in distinguishing between uncopyrightable and copyrightable elements of a computer program. *Lotus* v. *Paperback, 740* F. Supp. 37,62 (D. Mass. 1990).

120 For a discussion of ''idcast '' see Paul Goldstein, *Copyright Principles, Law and Practice* (Boston, MA: Little, Brown, 1989), vol. I, pp. 76-79.

protected design. There is a concern that a‘ ‘building block” could be appropriated through copyright by the developer of the first program to use it.

One question is related to the role of user interface design principles in determining the scope of design choices. One of these principles is that interface designers should be aware of the benefits of ‘external consistency. [121] External consistency allows the ‘‘transfer of learning’ from one program to another and from one generation of a program to the next. Is it necessary for two spreadsheet programs to be identical in virtually every respect, or can there be significant transfer of learning if two spreadsheet programs share only some core similarity? Would this core similarity be viewed as an unprotected ‘‘idea’ in the context of copyright law? At the same time, there is a concern that intellectual property law will force “gratuitous” differences between interfaces. [122]

Program Code

How Is the Program Code Protected?

The copying of a computer program can be prevented in several ways. If a computer program is the implementation of a patented process, then copying the program and practicing the invention would infringe the patent. Copying could also be limited by a licensing agreement between the developer of the software and a licensee. However, the main vehicle for preventing the copying of the program has been copyright law. Computer programs have been copyrightable subject matter since 1978, when the Copyright Act of 1976 became fully effective. [123]

The Copyright Act states that copyright protection does not extend to the “procedure’ or “system’ or ‘‘method of operation’ described by a copyrighted work. This is to prevent copyright from being used to protect ‘‘utilitarian’ or ‘‘fictional’ articles. For example, an electronic circuit is not copyrightable subject matter, but the circuit diagram that describes the circuit is a copyrightable “pictorial’ work. The copyright only prevents someone from copying the pictorial work, not from building the circuit. In the case of computer programs, it is especially difficult to separate the description of the function from the function itself.

“Idea” is a metaphor used in copyright law for the elements of a work that copyright law does not protect. Procedures, systems and methods of operation are ideas. ‘‘Expression ‘‘ is a metaphor for the protected elements of a work. Infringement occurs under copyright law when a work is copied and, taken together, the elements copied amount to an improper appropriation of expression. Copying can be shown by direct evidence or by inference, if the defendant had access to the plaintiff’s work and the works have substantial similarity as to the protected expression. Improper appropriation is shown by the taking of a substantial amount of protected ‘ ‘expression.

Literal Code—The literal code of a program has consistently been shown to be protected expression, and verbatim copying a copyright infringement. As a result, copying a program from one disk to another clearly infringes the copyright in the program, except to the extent permitted by the Copyright Act (e.g., section 117). This is true regardless of the language used to write the program: the argument that a program in executable (machine language) form was not copyrightable subject matter because it could be considered a ‘ ‘machine part’ has been rejected by the courts.

“Nonliteral” Copying—In a series of cases, courts have held that the internal design of a program at a level of abstraction above that of the program code could not be copied, In one case a judge wrote:

> It would probably be a violation to take a detailed description of a particular problem solution, such as a flowchart or step-by-step set of prose instructions, written in human language, and program such a description in computer language.[124]

In other words, a finding of infringement could not be avoided by making small changes to a program or

---

121 See pp. 129-130.

122 Participant in discussion at the Massachusetts Institute of Technology on “Intellectual Property in Computing: (How) Should Software Be Protected” (Cambridge, MA: Transcript, Oct. 30, 1990), p. 21.

123 House Report 94-1476 says ‘‘liteW works’’ protected under section 102(a)(1) of the Copyright Act include computer programs. The protection of computer programs under the Copyright Act was confirmed by the Software Amendments of 1980.

124 *Synercom* v. *UCC*, 462 F. Supp. 1003, 1013 n.5 (N.D. Tex. 1978).

by translating the program from one language to another language.[125]

The higher levels of abstraction of the program code are often described as the ''structure, sequence, and organization' [126] (SS0) of the program, although this terminology has been criticized,[127] Protection of the structure, sequence, and organization has been described as consistent with the application of copyright to more traditional types of ' 'literary' works such as novels.[128] The main reason to limit copying at this higher level of abstraction is that it would otherwise be possible to avoid copyright infringement by making a few trivial changes to the program text. The courts have determined that this would allow the appropriation of a significant part of the value of a program.[129]

There were two seminal cases in the area of protection for the structure, sequence, and organization of computer programs: *Whelan v. Jaslow* and *SAS v. S&H.* In these cases the particular organization of the program into subroutines or modules was found to be protected expression. In *SAS v. S&H* the court stated that copying the organizational scheme of a program would be a taking of expression, even if the program code for the ''lowest level tasks' were written independently, [130] In *Whelan v. Jaslow* the two programs were found to be substantially similar because of similarities in the detailed structure of the five subroutines that the court found to be qualitatively important to the program and ' 'virtually identical file structures. [171]

Constraints on Program Structure-The courts have applied the "merger" doctrine of copyright law by looking for evidence that the structure of the program was dictated by engineering constraints. In cases such as Q-Co. v. *Hoffman* and *NEC v. Intel the courts* have found that the similarities between programs were due to constraints imposed by the overall purpose of the program or by the hardware. If there were different ways of writing the program to perform a particular function, however, the courts have found protected expression. In *SAS v. S&H,* for example, the court wrote that:

> [The defendants] presented no evidence that the functional abilities, ideas, methods, and processes of SAS could be expressed in only very limited ways. *[32]

The number of different ways of writing a program to perform a particular function was discussed at hearings conducted by the National Commission on New Technological Uses of Copyrighted Works (CONTU).[133]

Copies—Because computer programs are protected by copyright, the making of *any* copy is an infringement. Even the transfer of a program from disk to memory is thought to be the creation of a copy that would be infringing but for the special exemption contained in section 117 of the Copyright Act, which allows a computer program to be copied ''as an essential step in the utilization of the computer program in conjunction with a machine.

The exclusive rights granted to the copyright holder are also thought by some to limit disassembly or recompilation of programs-these procedures involve the making of reproductions or ''derivative works' of the machine language program. Limitations on disassembly and recompilation provide trade secret protection for aspects of a program

125 See *Whelan* v. *Jaslow* 797 F.2d 1222 (3d Cir. 1986) and *SAS v. S&H,* 605 F. Supp. 816 (M.D. Tenn. 1985).

126 *Whelan* v. *Jaslow, 797* F.2d 1222, 1224 (3d Cir. 1986).

127 *Computer Associates* v, *Altai, No.* CV 89-0811, U.S. District Court, E.D. New York, Aug. 9,1991.

128 ''As I have indicated, CONTU had no views, and made no recommendations which would negate the availability of copyright protection for the detailed design, structure, and flow of a program under the copyright principles that make copyright protection available, in appropriate circumstances, for the structure and flow of a novel, a play or a motion picture. ' Declaration of Melville B. Nirnmer (Vice Chairman of CONTU), appendix to Anthony L. Clapes, Patrick Lynch, and Mark R. Steinberg, ''Silicon Epics and Binary Bards: Determining the Proper Scope of Copyright Protection for Computer programs," *UCLA Law Review, vol.* 34, June-August 1987, p. 1493.

129 ''. . .among the more significant costs in computer programming arc those attributable to developing the structure and logic of the program. The rule proposed here, which allows copyright protection beyond the literal code, would provide the proper incentive for programmers by protecting their most valuable efforts, while not giving them a stranglehold over the development of new computer devices that accomplish the same end. *Whelan* v. *Jaslow, 797* F.2d 1222, 1237 (3d Cir. 1986).

130 605 F. Supp 816, 826 (M.D. Tenn. 1985).

131 *Whelan* v, *Jaslow,* 797 F.2d 1222, 1228 (3d Cir. 1986).

132 605 F, Supp 816, 825 (M.D. Tcm. 1985).

133 See transcript of CONTU Meeting No. 10, pp. 44-45, quoted in declaration of Melville B, Nimmer, appendix to Clapes et. al., op. cit., footnote 128, p. 1588.

because the machine language version of programs is difficult to understand, The recompilation issue is discussed in more detail in a later section of this chapter.

Policy Issues-Protection of Literal and Nonliteral Elements of Program Code

Literal Copying—The justification for restrictions on the copying of computer programs is economic: some form of legal protection is necessary to provide program developers with the incentives to develop software. Computer programs are easy to copy—they have the same intangible character as traditionally copyrightable works. The CONTU Final Report states:

> The Commission is, therefore, satisfied that some form of protection is necessary to encourage the creation and broad distribution of computer programs in a competitive market,[134]

The Commission viewed computer program copyrightability as consistent with the expansion of copyright to new technologies over the previous two centuries, [135]

Protected and Unprotected Elements of Program Code—One of the reasons for protecting nonliteral elements of a program is to prevent later developers from avoiding a finding of infringement by making small changes. The main issue is the degree of similarity two programs may have and the degree of independent development that a later developer will be forced to do. In other words, to what extent can the intellectual work in one program be used in a second program? In the two important structure, sequence, and organization cases, *Whelan v. Jaslow* and *SAS v. S&H, the infringing* program's code was similar at a low level of detail. The line drawn between protected and unprotected elements reflects a determination of the level of competition desirable. [136]rticulating this line has proven to be difficult.

In practice, there will rarely be access to the high-level language version of a competitor's program. The only access that one would normally have to a competitor's program would be to its machine language form. Disassembly would be possible, but there would still be considerable work involved in understanding the program and reimplementing it. It is not surprising that in the structure, sequence, and organization cases there has either been access to the source code or the programs were short enough to be disassembled and studied relatively easily.[137] The legal status of attempts to disassemble a program is a major issue associated with the protection of computer programs using copyright law, and is the subject of the next section.

Recompilation

''Recompilation' is a procedure for translating a machine language program into a more understandable form. It is thought by some to be a copyright infringement, and by others to be a necessary tool for software engineering. The recompilation issue intersects many of the policy issues outlined earlier in this chapter. For example, recompilation may be used in the development of functionally compatible products; whether or not the development of such products should be permitted is itself a policy issue (see ''External Design—Policy Issues" in this chapter).

Introduction

To understand a program, there are three things you can do: read about it (e.g., documentation), read it (e. g., source code), or run it (e.g., watch execution, get trace data, examine dynamic storage, etc.).[138] Understanding a program is made easier when the high-level language or assembly language representations are available. In most cases, however, only the machine language version is distributed. Decompilation is a procedure by which a high-level language representation of a program is derived from a machine language program, and ' 'disassembly" is a procedure for translating the machine language program into an assembly language program.

---

134 National Commission on New Technological Uses of Copyrighted Works (CONTU), *Final Report* (Washington DC: Library of Congress, July 31, 1978), p. 11.

135 Ibid.

136 ''LaST Frontier Conference Report on Copyright Protection of Computer Software, " *Jurimetrics*, vol. 30, No. 1, fall 1989, p. 20.

137 *E.F. Johnson v. Uniden*, 623 F.Supp. 1485 (D.Minn. 1985) (a radio communications product), *NEC v. Intel 10* U. S. P.Q.2d 1177 (N.D. Cal. 1989) (microcode).

138 Richard B. Butler and Thomas A. Corbi, "Program Understanding: Challenge for the 1990' s," *Scaling Up: A Research Agenda for Software Engineering* (Washington, DC: National Academy Press, 1989), p. 41.

The legal status of efforts to discover assembly language or high-level language representations of a program has become the subject of an intense debate.[139] Both recompilation and disassembly involve the making of at least a partial reproduction or derivative of the machine language program, and some people believe that reverse engineering using these techniques is a copyright infringement.[140] The policy question is the extent to which limiting access to information about someone else's program through the workings of the copyright law is socially desirable.

Important factors cited in the policy debate are the uses of recompilation, the ease with which it can be done, and the degree to which the information is available from sources other than recompilation. It has been argued that limits placed on recompilation are required to provide sufficient incentives for the development of original programs. Those who take this position claim that recompilation is a straightforward and routine process that allows clone programs to be implemented at much lower cost,[141] Programs are decompiled and then:

> ... without the necessity for the significant R&D expenditures made by the innovator, the pirate goes on to alter the program to disguise the copying, and create a second, similar program which it markets as an allegedly different product for a much lower price. '[42]

Others argue that recompilation is technically difficult, and is therefore unlikely to be used for piracy. They emphasize that disassembly and recompilation can be used for a variety of other purposes, many of which would have a less direct economic impact on the developer of the program being reverse engineered. For example, some of the information gained by recompilation may be used in developing an "attaching' product that is to exchange data with the program being reverse engineered. Recompilation also could be used for maintenance, debugging, detecting viruses, investigating safety or reliability concerns, or systems integration. Indeed, some of these uses of decompilation represent situations in which an organization might reverse engineer its own programs, not just those developed by someone else.[143]

Recompilation and Disassembly

The product of recompilation or disassembly would never be identical to the original source program.[144] At the very least, comments and the names of labels, variables, and procedures would be lost in the assembly or compilation process and could not be recovered. In addition, the structure of the decompiled program would not necessarily be the same as that of the original program, although this would depend on the compiler that had been used. Because of the loss of mnemonics and much of the structure of the program, considerable work is required to understand the decompiled or disassembled program.

Disassembly is easier than recompilation. There is essentially a one-to-one conversion between the machine language statements and assembly language statements, simplifying the process of translating the machine language program into a more readable form. However, it takes a great deal of effort to understand the disassembled code from a large program.[145] Because disassemblers are **widely** available,[146] some developers assume that their programs will be disassembled, and try to write sensitive parts of their code in ways which make disassembly more difficult or make the disassem-

---

139 Pamela Samuelson, "Reverse-Engineering Someone Else's Software: Is It Legal?" *IEEE Software, vol. 7, No.* 1, January 1990, pp. 90-96.

140 Victor Siber, Corporate Counsel, IBM Corp., "Interpreting Reverse Engineering Law, ' letter to *IEEE Software,* vol. 7, No. 4, July 1990, p. 8.

141 "Decompilation of a computer program does not provide an imitator with just a good start in producing a competing product; it gives him virtually everything necessary to produce a functionally identical product, ' William T. Lake, John H. Harwood, and Thomas P. Olson, "Tampering With Fundamentals: A Critique of Proposed Changes in EC Software Protection,' *The Computer Lawyer*, vol. 6, No. 12, December 1989, pp. 1-10.

142 Testimony of James M. Burger, Chief Counsel, Apple Computer, Inc., on behalf Of the Computer and Business Equipment Manufacturers Association, at hearings before the House Subcommittee on Intellectual Property and Judicial Administration, May 30, 1991.

143 This situation presents no infringement issues.

144 The lack of identity is not relevant to the legal question of unauthorized copying.

145 An 80,000-byte machine language program for an IBM PC-class computer would result in 32,000 lines of *assembly code.* Clark Calkins, ' 'Tailoring the MD86 Disassembler for Turbo Pascal,' *Tech Specialism, vol. 2, No. 6,* June 1991, pp. 41-46.

146 For a discussion of commercially available disassemblers, see Brett Glass, ' 'Disassembler Roundup, '' *Programmer' s Journal, vol. 9.2,* March/April 1991, pp. 66-71.

bled code more difficult to understand.[147] Figure 4-2 shows a high-level language program, the corresponding machine language (compiled) program, and the results of disassembling the machine language program.

Recompilation is much more difficult; at this time there appear to be no commercially available decompilers. For this reason, it is unclear whether decompilation is widely used by ‘ ‘pirates” to decompile entire programs and then rearrange the code in an attempt to hide the copying,[148] It is possible that the term “recompilation ‘‘ is being used in the policy debate to include disassembly,[149] as “decompilation’ is often characterized as any technique that is used to transform ‘‘machine readable’ code into ‘‘human readable’ code.[150]’

Today, any effort to decompile a program would start with disassembly. Then, if one knew something about the compiler that had been used, it might be possible to match certain patterns of assembly language statements to higher level constructs. However, recompilation would be much more difficult in cases where a sophisticated compiler had been used: *optimizing* compilers delete and rearrange some of the instructions in order to make the machine language program more efficient, and the correspondence between sequences of machine language instructions and high-level instructions becomes less direct. While a pseudo-source code program could still be derived, it would be less likely that the decompiled program would immediately reveal the original program structure.

Uses of Recompilation

The information gained by reverse engineering techniques such as recompilation can be put to a variety of uses, each with a different economic effect. The effect on the developer of the program being decompiled is most direct when the information is being used to develop a competitive product, In some cases the reverse engineer maybe interested in learning about a small part of a program, such as an algorithm, that gives the program’s owner a competitive advantage.[151] In other cases, decompilation could be used to develop the specifications for a program that is fictionally compatible—a clone program, Sometimes the specifications are used in a clean-room process that is intended to ensure that the new program does not share expression with the original, for it is the protected expression that is protected by copyright.[152]

Recompilation may also be used to develop a program or hardware device that is not competitive, but “complementary” or “attaching.” This would not affect the market for the original product directly, but would create more competition in the second market. For example, knowing interface information might allow the development of competition in the market for peripheral devices such as printers. Recompilation can also be used to confirm published interface specifications; for example, in the course of debugging a program an unexpected problem may arise with another program in the system, such as an operating system.

Finally, there are a variety of uses for decompilation for which no product is developed at all. First of

---

147 Bob Edgar, “Shielded Code: How To Protect Your Proprietary Code From Disassemblers, ” *Computer Language*, vol. *8, No. 6,* June 1991, pp. 65-71.

148 Following the testimony of the computer and Business Equipment Manufacturers Association at the May 30,1991 hearings (See footnote 142), OTA asked for specific examples of piracy using recompilation and descriptions of the state of the art in automated recompilation (Joan Winston, OTA, letters to James Burger, Apple Computer, July 5, 1991 and Sept. 23, 1991). To date, OTA has not been provided with this information.

149 ‘‘A computer program is generally written, in the first instance, in ‘source code’ —that is, in a relatively high-level language such as FORTRAN or Pascal. The program is then translated (or ‘compiled’) into ‘object code, ’ which consists of instructions to the computer in the form of O’s and 1‘s. Programs are frequently distributed to customers only in object code form; the source code is retained as an unpublished, copyrighted work. Recompilation and disassembly (*which* we calf *‘decompilation’ for short*) *are* methods of reconstructing the source code of a program through copying and manipulation of its object code. ’ Lake et al., op. cit., footnote 141, p. 4 [emphasis added].

150 The legal issue is the same, whether a program is “disassembled” or “decompiled.”

151 ‘‘The source code, which often contains the trade secrets of the software creator, remains unpublished. Many software companies go to great lengths to keep their proprietary source codes confidential. . . The right to decide not to publish in any form source code goes to the heart of most software companies’ strategies for retaining the confidentiality of their most valuable and carefully guarded trade secrets. William Neukom, Vice President, Law and Corporate Affairs, Microsoft Corp., on behalf of the Software Publishers Association at hearings before the House Subcommittee on Intellectual Property and Judicial Administration, May 30, 1991.

152 For discussion of clean room issues, see David L. Hayes, ‘‘Acquiring and Protecting Technology: The Intellectual Property Audit,’’ *The Computer Lawyer,* vol. *8, No. 4,* April 1991, pp. 1-20. The effectiveness of using a clean room to avoid copyright infringement depends on whether the specifications that are used are ideas and not expression.

Figure 4-2-High-Level Language, Machine Language, and Disassembled Versions of a Program

HIGH-LEVEL LANGUAGE PROGRAM

```
program sum–of–numbers;
{This program adds the numbers from first–number to last–number}
{The text between curly brackets is known as a "comment." Comments }
{make a program easier to read and understand, but do not affect the }
{execution of the program.                                           }
var first–number, last–number, i, sum: integer;
begin
        {initialize variables}
        first–number := 1,
        last–number := 5;
        sum :_O;
        {add numbers from first–number to last–number}
        for i := first–number to last–number do
        begin
                sum := sum + i;
        end;
        {print the sum}
        writeln('The sum is sum);
end.
```

The program shown above, written in the high-level language Pascal, adds the numbers from 1 to 5. High-level language programs have to be translated (compiled) into machine language in order to be executed on the computer. Part of the compiled program is shown below.

MACHINE LANGUAGE (COMPILED) PROGRAM

```
10111000 00000001 00000000 10100011 01100000 00000010 10111000 00000101
00000000 10100011 01100010 00000010 10111000 00000000 00000000 10100011
01100110 00000010 10100001 01100000 00000010 01010000 01000001 01100010
00000010 01011001 10010001 00101011 11001000 01111101 00000011 11101001
00011010 00000000 01000001 10100011 01100100 00000010 01010001 10100001
01100110 00000010 00000011 00000110 01100100 00000010 10100011 01100110
00000010 01011001 00101001 01110100 00000111 11111111 00000110 01100100
00000010 11101001 11101010 11111111 11101000 01111010 11110111 11101000
01111100
```

Machine language programs are difficult to read and understand. If the original high-level language program is not available, disassembler programs may be used to translate the machine language program into a more understandable form called assembly language. However, assembly language programs are still more difficult to understand than high-level language programs. Part of the assembly language program is shown below.

DISASSEMBLED PROGRAM

```
2D9F  MOV   Ax, 0001          2DC1  JMP   2DDB
      MOV   [0260], AX              INC   CX
      MOV   Ax, 0005          2DC5  MOV   [0264], AX
      MOV   [0262], AX              Push  CX
      MOV   Ax, 0000                MOV   AX, [0266]
      MOV   [0266], AX              ADD   AX, [0264]
      MOV   AX, [0260]              MOV   [0266], AX
      PUSH  Ax                      POP   CX
      MOV   Ax [0262]               DEC   CX
      POP                           JZ    2DDB
      XCHG  Cx, Ax                  INC   WORD PTR[0264]
      SUB   Cx, Ax                  JMP   2DC5
      JGE   2DC1              2DDB  CALL  2558
```

SOURCE: OTA.

all, someone may wish to ''maintain' a program for which no source code is available. Recompilation or disassembly would help in understanding the program so that it could be adapted to new requirements, or to fix bugs if no other support was available. Disassembly is also used to find viruses, to examine the output of a compiler to see what it had done, and finally to examine a competitor's program to see if they had taken any protected expression.

Other Methods of Reverse Engineering

There are other methods of reverse engineering programs whose legal status is less controversial because they do not involve the making of unauthorized reproductions of the machine language program.[153] In practice, a reverse engineer would probably employ a combination of methods, depending on the application, the information being sought, the effort involved, and legal considerations. Some information can be obtained by simply executing the program: it can be run with many different data sets and its behavior observed. There are a number of different software and hardware tools that could be used to follow the course of execution of the program. However, the program code is the best specification of the behavior of the program-it may be impossible to develop tests to explore all the cases that a program may have to handle.

Other information is available from published specifications, manuals, and standards documents. In some cases companies will publish interface specifications because it is in their commercial interest to do so. Even if the information is not published, they may be willing to make it available through contractual arrangements. However, in other cases, such as when a company is active in the market for both the primary product and a complementary product, it may want to limit competition in the secondary market by not making the interface information available. Published documents may not be at the appropriate level of detail. For example, there may be scope for differences between implementations of a standard and manuals may be inaccurate or out of date, or leave some elements undocumented.

Legal Arguments for Policy Positions

While there have been proposals that a new ***sui generis*** law be enacted to protect software, much of the discussion of software intellectual property policy issues has been based on interpretations of current law. Convincing legal arguments have been made for many of the policy positions discussed in the preceding sections. The two broadest legal questions are the proper interpretation of the ''mental steps" and "law of nature" exceptions to patentability in patent law, and the proper interpretation of the statement in section 102(b) of the Copyright Act that copyright protection does not extend to ''processes' or ''methods of operation. Both the exceptions to patentability and the meaning of section 102(b) have been given a number of different interpretations by legal scholars and the courts.

Patent Law

One, policy position is that inventions implemented in software should not be statutory subject matter. It has been argued that the "mental steps" doctrine can be used to exclude software implementations from the patent system.[154] Under this doctrine, processes that could be performed using pencil and paper are not statutory. The U.S. Supreme Court in its *Benson* opinion wrote that a computer does arithmetic ''as a person would do it by head and hand. ''155 In the late 1960s PTO used the mental steps doctrine to deny patents to inventions that used software.

The view that inventions that use software are only statutory if they are traditional industrial processes that transform matter may also be supported by the case law. In *Benson, the* Supreme Court, relying on a series of cases from the 1800s, wrote that "[t]ransformation and reduction of an article 'to a different state or thing' is the clue to patentability of a process claim. ''[156] However, the Court did goon to say that it was not holding that no

153 ''. . . the ideas and principles underlying a program ean frequently be discovered in other ways---ways that are legitimate. Examples are studying published documentation, performing timing tests and observing the inputs, outputs, and conditions of operation," Victor Siber, op. cit., footnote 140, p. 8.

154 Pamela Samuelson, "Benson Revisited, " *Emory Low Journal*, vol. 39, No. 4, fall 1990, p. 1047-1048.

155 *Gottschalk* v. *Benson,* 93 S. Ct. 253,254.

156 93 S. Ct. 253,256.

process patent could ever qualify if it did not operate to change articles or materials.[157]

Legal arguments can also be used to support the position that some of what are now deemed nonstatutory mathematical algorithms should be patentable. These arguments are based on the fact that the Supreme Court appeared to view the *Benson algo*rithm as a ‘‘law of nature. ’ Some have argued that the Benson algorithm was not the mathematical expression of a scientific truth, such as F=ma expresses the relationship between force, mass, and acceleration, but a man-made solution to a complex problem.[158] According to this interpretation of patent law, industrially useful processes should not become unpatentable merely because they can be described mathematically.[159]

Copyright Law

The scope of copyright protection for computer programs depends in part on the interpretation of the meaning of section 102(b) of the Copyright Act. With all works courts must engage in the process of drawing the line between protectable expression and unprotectable ‘‘ idea[s], procedures], processes], system[s], method[s] of operation, concepts], principles], or discoveries].’’[160] This e$_{xerc}$ise becomes more critical, and difficult, in the context of fact-based works, such as history texts and instruction manuals, and ‘‘functional’ works, such as blueprints or computer programs,

There are a number of different views of the application of existing law to user interfaces. One interpretation of the law is that user interfaces are inherently functional and therefore not copyrightable subject matter. According to this interpretation, user interfaces are in the domain of patent law,[161] protected only to the extent that elements are novel and nonobvious. This argument would support a policy position that sharply limits the scope of protection for user interfaces.

The other view is that user interfaces may be protected by copyright. One approach has been to protect the user interface screen displays as audiovisual works or compilations of literary terms.[162] The screen displays are considered a separate work from the program code. As for all works, the scope of protection for the audiovisual work or compilation is determined by an interpretation of section 102(b). One interpretation is that the command terms are ‘ideas’ and that only their arrangement on the screen is protected expression.[163] Protection of the command terms themselves can be supported by an interpretation of section 102@) in which the unprotected ‘ ‘idea’ is at a higher level of abstraction, such as the overall purpose of the program. Any design choices not necessary to the purpose of the program, including the choice of command terms, would then be protected expression.

A second approach to protecting user interfaces through copyright law is to consider the user interface as protected by the copyright in the program. The user interface is viewed as part of the ‘‘structure, sequence, and organization’ of the underlying program.[164] This arguably represents a different interpretation of the meaning of structure, sequence, and organization’ from the way in which the term was used in *Whelan:* `it` is possible to create two programs that have identical user interfaces but use different subroutines and data structures (the elements that contributed to the court’s finding of similarity of SS0 in *Whelan).*

The term ‘structure, sequence, and organization’ has been criticized for failing to distinguish between the ‘‘static’ structure of the program-the program code—and its ‘‘dynamic’ structure-the ‘ ‘behavior’ of the program when loaded into the computer

15793 S. Ct. 253,257.

158 Laurie, op.cit., footnote 61, p.257.

159 William L, Keefauver, ‘ ‘The Outer Limits of Software Patents, in Morgan Chu and Ronald S. Laurie (eds.), *Patent Protection for Computer Software* (Englewood Cliffs, NJ: Prentice Hall Law and Business, 1991), p. 83.

160 17 U.S.C. 102(b).

161 Steven M, Lundberg, Michelle M, Michel, and John P. Sumner, ‘ ‘The Copyright/Patent Interface: Why Utilitarian ‘Look and Feel’ Is Uncopyrightable Subject Matter, ” *The Computer Lawyer, vol. 6, No.* 1, January 1989.

162 *Digital Communications Associates* v. *Softklone*, 659 F. Supp. 449 (N.D. Ga. 1987).

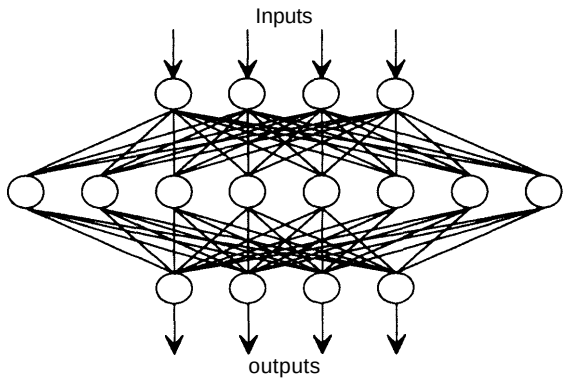
163 Ibid.

164 *Lotus* v. *Paperback, 740* F. Supp. 37, 80 (D. Mass. 1990).

Box 4-C—Neural Networks

Neural networks are a special kind of computing architecture.[1] The network consists of a large number of interconnected processing elements, arranged in layers (see figure 4-C-l). The relationship between the input and output of the network is determined by the internal details of the network. Signals passing between the layers of the network are modi.tied by multiplying them by "weights." These weighted signals are received as inputs by the processing elements. The processors compute an output value, which is a function of the sum of the inputs, and passes the output to the next layer of processors.

Figure 4-C-1—Neural Network



The circles represent processing elements that perform a weighted **sum on their inputs and compute an output value that is then sent to the next layer of processing elements.**

SOURCE: OTA.

The weights determine the overall behavior of the network, much as the program code determines the behavior of a conventional computer. However, neural networks are not programmed in the same way as conventional computers. Neural networks are ''trained. The network is presented with input values for which the desired output is known. The network then adjusts the weights until this output is achieved. This process is repeated for a large number of input and output examples, called a training set. Given enough examples, the desired behavior of the network can be achieved for a wide range of inputs. One focus of research on neural network applications has been "pattern recognition" problems such as recognizing handwritten characters: the input is image data, and the output indicates which letter has been "read."[2]

Because neural networks are different from conventional computers, there is some uncertainty about the application of intellectual property laws.[3] One issue is the copyrightability of the set of weights. For example, do the weights satisfy the Copyright Act's definition of a computer program? Can the set of weights be said to be a work of authorship? One could argue that the network, and not a human, actually authors the weights (see box 4-A on authorship). On the other hand, the network could be regarded simply as a tool used by a human author-the author chooses the training set and presents the data to the network. The first copyright registration for a set of neural network weights issued in October 1990.

A second question is whether protecting the weights alone is sufficient to protect the value embodied by the network. Much as two programs can have the same external design or input/output relationship but different program code, two networks can have the same input/output relationship but different sets of weights. The ability of a neural network to "learn" could make it easier to appropriate the value of the network—the input/output relationship-without actually copying the weights. An existing network or conventional program could be supplied with inputs and the outputs observed.[4] These sets of input and output data could then be used to train a second network, which would have similar behavior to the original network.

---

1 For an introduction to neural networks, see Judith Dayhoff, *Neural Network Achitectures* (New York, NY: Van Nostrand Reinhold, 1990).

2 Tim Studt, ''Neural Networks: Computer Toolbox for the 90's,'' *R&D Magazine*, vol. 33, No. 10, September 1991, p. 36.

3 Andy Johnson-Laird, "Neural Networks: The Next Intellectual Property Nightmare?" *Computer Lawyer*, vol. 7, No. 3, March 1990, p. 7; Gerald H. Robinson, "Protection of Intellectual Property Protection in Neural Networks," *Computer Lawyer*, vol. 7, No. 3, March 1990, p. 17; Donald L. Wenskay, ''Neural Networks: A Prescription for Effective Protection' *Computer Lawyer, vol. 8, No. 6,* August 1991, p. 12.

4 Johnson-Laird, op. cit., footnote 3, pp. 14-15.

SOURCE: OTA and cited sources.

and executed.[165] The possibility that the behavior of a program could be protected expression led to a discussion about the extent to which copyright protection might overlap with patent protection of the program function—what some have termed the "patent/copyright interface" problem.[165] The relationship between intellectual property protection of static and dynamic structure is also an issue in the context of neural networks (see box 4-C).

The same issue was addressed by the Copyright Office in 1988 when it addressed the nature of the relationship between the program code and screen displays. Hearings were held by the Office in response to the *Softklone court's* holding that the ''computer program' copyright did not extend to the screen displays. The *Softklone* court had noted that "the same screen can be created by a variety of separate and independent computer programs. [167] At the hearings the IEEE Computer Society supported separate registrations of the program code and screen displays, arguing that the nature of the ''authorship '' in the program code was fundamentally different from that in the screen displays.[168] However, the Copyright Office ruled that a single registration of a computer program covered any copyrightable authorship in the program code and the screen displays, writing that ''the computer program code and screen displays are integrally related and ordinarily form a single work. ''[169]

Software Development

Arguments about the proper interpretation of existing law also rely in part on characterizations of the software development process. Some emphasize ''creative' aspects of the development process. Just as with other copyrightable works, it is argued, this creative effort should be encouraged by limitations on copying, Others, however, characterize the development process as ''engineering, ' in an effort to limit the scope of copyright protection or to argue for the wider use of patents.[170]

Discussions of creativity and engineering can also be seen as related to the scope of available design choices. One of the goals of software engineering methodologies is to reduce the number of design decisions, as a way of managing the complexity of large projects. Elements of the development process have become more routine. High-level languages free programs from much of what Brooks calls "accidental complexity. ' '[171] Shaw points out that today "almost nobody believes that new kinds of loops should be invented as a routine practice. ' '[172] Subroutines, macros, and operating systems have also been used to avoid ''re-inventing the wheel. " The concept of reuse (see box 4-D) may also make parts of the development process more routine. The Federal Government, particularly the Department of Defense, has shown considerable interest in encouraging reuse (see box 4-E).

165 ''Ccn@d to Dr. Davis criticism of the Whelan 'structure, sequence, and organization' formulation is the fact that there is no necessary relationship between the sequence of operations in a program, which arc part of behavior, and the order or sequence in which these operations arc set forth in the text of the program—the source code and object code. As Dr. Davis pointed out, 'the order in which sub-routines appear in the program text is utterly irrelevant, ' and the two views of a computer program, as text and as behavior, arc 'quite distinct. ' '' *Computer Associates v. Altai*, op. cit., footnote 127, p. 14.

166 See Pamela Samuelson, "Survey on the Patent/Copyright Interface for Computer Programs, " *AIPLA QJ. vol.* **17,** p. **256.** See **also** *Computer Associates v. Altai*, op. cit., footnote 127, p. 15. A study prepared jointfy by the Patent and Trademark Office and the Copyright Office concluded that there is minimal overlap between the two areas with respect to computer software. U. S. Patent and Trademark Office and U. S. Copyright Office, *Patent-Cop yright Laws Overlap Study*, May 1991, pp. ii-iii.

167 *Digital Communications* Associates v. *Softklone* Distributing, 659 F. Supp. 449, 455-456. The court then concluded, ' 'Therefore, it is the court's opinion that a computer program's copyright protection docs not extend to the program's screen displays, and that copying of a program's screen displays, without evidence of copying of the program's source code, object code, sequence, organization, or structure does not state a claim of infringement.

168 Richard H. Stem, "Appropriate and Inappropriate Legal Protection of User Interfaces and Screen Displays, Part l," *IEEE Micro, vol. 9, No. 3,* June 1989, p. 84.

169 Copyright Office, ''Registration Decision; Registration and Deposit of Computer Screen Display s,' 53 *Federal Register* 21819 (June 10, 1988).

170 For one view of the relationship between ''software engineering" and intellectual property, see Clapes, op. cit., footnote 96, pp. 119-120. For extensive discussion of the nature of software development, see Susan Lammers, *Programmers at Work* (Redmond, WA: Microsoft Press, 1986).

171 Frederick p, Brooks, Jr., ' 'No Silver Bullet, ' *IEEE Computer*, April 1987, *p.* 12.

172 Mary Shaw, "Prospects for an Engineering Discipline of Software, ' *IEEE Software*, vol. 7, No. 6, November 1990, p. 22.

Box 4-D—Software Reuse

Productivity in software development is a concern in both the private and public sectors.[1] The relatively low productivity of software programmers is a difficult problem, so one way to improve programming productivity is to "reuse" program code.[2] This would eliminate much of the redundant work of many programmers writing code that does essentially the same thing. One source estimated that of 15.3 billion lines of code written in 1990, only 30 to 40 percent represent novel applications; 60 to 70 percent represent generic computer tasks like data entry, storage, and sorting.[3]

Reuse can be either accidental or systematic. Many programmers employ "accidental' reuse, making use of some elements of their own previous work or that of their colleagues. In systematic reuse, software is written from the beginning with the intention of making it more reusable; the components are documented and put in a library. This can be time consuming, and in the short run can be more costly than writing a specific program for the immediate need. This cost has to be seen as an investment that pays off in the long run if the component can be reused several times. Software development to facilitate systematic reuse could also streamline software maintenance, which accounts for a large and increasing portion of software life-cycle costs.

When reuse is being practiced within an organization and programmers only use components from their own organization's software library, intellectual property considerations are not an issue. However, questions of ownership become more important if there is to be development of a market in reusable components. This market is growing, but is still relatively small. For example, it is possible to license libraries of code for common functions, such as components used in developing graphical user interfaces.

Intellectual property considerations can affect reuse in three ways. First, a number of participants at an OTA workshop on software engineering indicated that uncertainty about the ownership of a component or the scope of intellectual property rights could discourage the development of programs composed of components from different sources. Second, some in the reuse community think that a stable system of ownership rights[4] is necessary to encourage the investment required for creating a commercial-quality library and to handle questions of liability. In some cases in the past, the investment for widely used libraries has come from sources other than potential licensing fees (e.g., the X-windows library developed in a university research and education environment, at MIT, and later used in commercial products). A final issue is whether the interfaces in libraries of reusable components are protected by copyright law: can a competitor offer a library with the same interfaces but different implementations?[6]

There are a number of' other factors which affect the degree of reuse:[7]

1. Development standards have not been established for software;
2. There is a pervasive belief that if it is "not developed here," it can't be trusted or used by "us";
3. Software is all too often developed with respect to a specific requirement with no consideration given to reuse in other environments;
4. Many languages encourage constructs that are not conducive to reuse;

---

1 See, e.g., *The Software Challenge* (Alexandria, VA: Time-Life Books, 1988); and Albert F. Case, Jr., *Information Systems Development: Principles of Computer Aided Software Engineering (New* York, NY: Prentice-Hall, 1986).

2 See Kazuo Matsumura et al., "Trend Toward Reusable Module Component: Design and Coding Technique 50SM," in *Proceedings of the Eleventh Annual International Computer Software and Applications Conference—COMPSAC '87* (Washington, DC: IEEE Computer Society Press, Oct. 7-9, 1987), p. 45 (cited in Michael Cusumano, *Japan's Software Factories: A Challenge to U.S. Management (New* York, *NY:* Oxford University Press, 1991), p. 258).

3 David Eichmann and John Atkins, "Design of a Lattice-Based Faceted Classification System," paper presented at Software Engineering and Knowledge Engineering (SEKE '90), Skokie, IL, June 21-23, 1990.

4 Constructing licenses or pricing structures may be difficult. As with any digital information (see ch. 5), it will be hard to control what a user does with a component once a copy has been obtained. To&y, some libraries are being sold on a per-copy basis as source code with no royalties or runtime licenses, However some believe that market forces under the classical copyright paradigm-where "copies' are priced and sold—will not work properly (Brad Cox (Washington, CT), personal communication, Aug. 1, 1991). To reduce individual transaction costs, Cox suggests that use-base fees be administered collectively, similar to the way in which performance royalties for musical compositions are administered.

5 Robert W. Scheifler and James Gettys, *X Window System* (Bedford, MA: Digital press, 1990), pp. 8-15.

6 Institute for Defense Analyses, *Proceedings of the Workshop on Legal Issues in Software Reuse*, IDA Document D-1OO4 (Alexandria, VA: Institute for Defense Analyses, July 1991).

7 From Eichman and Atkins, op. cit., footnote 3.

5. Software engineering principles are not widely practiced and consequently, requirements and design documents often are not available with the code; and
6. No widely accepted methodology has been developed to facilitate the identification and access of reusable components.

There is an ongoing body of research designed to: 1) identify characteristics of software components that make them suitable for reuse, *2)* identify techniques to translate a software component with marginal reuse potential into one that can be reused, and 3) develop systems for classifying and identifying software components to make it easy to retrieve them from databases when they are needed.[8] Among the systems being considered are artificial intelligence programs capable of browsing libraries of programs, rating their qualit y according to several reusability y criteria (e.g., modularity, cohesion, size, control structure), and indicating those most suitable for reuse.[9] Much of this work is being done for, or in conjunction with, the Department of Defense (DOD), especially the Defense Advanced Research Projects Agency. As a major user of software, DOD has an interest in improving its own and its contractors' productivity through fostering reuse of software.

Reuse is more common among some major software users in Japan. Of several firms surveyed, Toshiba reported the most reuse with 50 percent of its delivered custom applications software being made of reused components.[10] Toshiba has made software reuse a central strategy for increasing productivity and reliability while reducing costs. Reuse is a high priority for both managers and programmers, Managers are rated on how well their projects have met reuse targets, as well as more usual measures like schedules and customer requirements. Programmers are required to report periodically on how many components they have used from, or contributed to, the reuse database; the company rewards authors of successful components that are frequently reused by others. Toshiba has also developed a specialized tool, OKBL (object-oriented knowledge-based language), which helps users classify components for storage in, or retrieval from, departmental libraries. Users can also locate components using printed catalogs. Most reuse, even at Toshiba, is within families of related products: less than 10 percent of software is reused across departmental lines.

8 Ibid. See also V.R. Basili, H.D. Rombach, J. Bailey, A. Delis, F. Farhat, "Ada Reuse Metrics," and R. Gagliano, G.S. Owen, M.D. Fraser, K.N. King, P.A. Honhanen, "Tools for Managing a Library of Reusable Ada Components," paper presented at Ada Reuse and Metrics Workshop, Atlanta, GA, June 15-16, 1988.

9 Jan Carlos Esteva and Robert G. Reynolds, "Learning To Recognize Reusable Software by Induction," paper presented at Software Engineering and Knowledge Engineering (SEKE '90), Skokie, IL, June 21-23, 1990.

10 Cusumano, op. cit., footnote 2, p. 261.

SOURCE: OTA and cited sources.

However, despite these advances, Brooks argues that part of software development will always be a creative process.[173] After reviewing the development of software engineering, he concluded that while the difference between poor conceptual designs and good ones may lie in the soundness of the design method (and cart be addressed by progress in software engineering), changes in methodology cannot bridge the gap between a good design and a great one. This, Brooks concluded, requires great designers.[174]

Debate continues within the field concerning the extent to which computer science should be characterized as a science or as an engineering discipline, its maturity as a discipline, and the appropriate content of undergraduate education in the discipline.[175] Some recent efforts have presented a formal definition of the discipline, its methodologies, and

173 Brooks, op. cit., footnote 171, p. 18.

174 Ibid.

175 For some recent discussion of these topics, see David Gries et al., "The 1988 Snowbird Report: A Discipline Matures," *Communications of the ACM*, vol. 32, No. 3, March 1989, pp. 294-297; Nor-man E. Gibbs, "The SEI Education Program The Challenge of Teaching Future Software Engineers," *Communications of the ACM, vol.* 32, *No.* 5, May 1989, pp 594-605; and Edsger W Dijkstra et al., "A Debate on Teaching Computer Science," *Communications of the ACM, vol.* 32, No. 12, December 1989, pp. 1397-1414.

Box 4-E—Special Concerns of the Federal Government

As a major user and developerof software, the Federal Govemment has special concerns with regard to future trends in software development. Due to the variety of missions of government agencies, its software needs span the gamut from small standard packages (word processing, spreadsheets, graphics) to large, specialized mission-critical systems (air traffic control, hospital information systems, military command and control) and nearly everything in between. Concerns include procurement policies, development of large composed systems, and technology transfer.

Procurement

Government procurement of computer hardware and software has been a complex and controversial subject for a long time. The government strategies for acquiring and managing information technology have been in a state of flux since passage of the Brooks Act of 1965,[1] which was enacted to establish procurement and management policies. Among concerns that have generated this flux are: 1) tension between the rapid pace of change in agency needs and improvements in technology versus the slow pace of the planning and procurement process; and 2) the tension between agency desires to ensure compatibility between systems and congressional desires to ensure competition among vendors.[2]

Software Development

Many government agencies are supported by software systems that are critical to performance of the agency's mission. These large systems, to be successful, require a good match between planning and assessment of technology needs and the acquisition or development of the hardware and software to match those needs.[3] In creating their systems, agencies face the choice of developing their software in-house, attempting to purchase ‘ ‘off-the-shelf packages to meet their needs, contracting with outsiders to develop customized software for them, or some combination of the three.

Once systems are in place, the complexities of the procurement process often ensure that they stay in place a long time. For these complicated systems, modifications and updates over the years make the software extremely complex and difficult to maintain. For example, the Social Security Administration's (SSA) system, in place since the early 1960s, had to be modified to reflect changes in benefits mandated by 15 laws passed between 1972 and 1981. Time allowed to make the changes was always inadequate, many mistakes were made, and backlogs became a recurrent problem, By 1982 the SSA faced the possibility of a ‘‘potential disruption of service' due to software deficiencies, yet by 1986 a system modernization program was still mired in political and legal problems and had barely begun.[4]

1 Public Law 89-306.

2 A detailed study of options for management of government information resources is found in U.S. Congress, office Of Technology Assessment, *Federal Government Information Technology: Management, Security, and Congressional Oveersight*, OTA-CIT-297 (Washington, DC: Government Printing Office), February 1986.

3 OTA has taken a close look at software development/proc urement problems at several agencies, including Federal Aviation Administration% Social Security Administration and Veterans Administration: U.S. Congress, Office of Technology Assessment, *Review of FAA's 1982 National Airspace System Plan,* OTA-STI-176 (Washington, DC: U.S. Government Printing Office, August 1982); *The Social Security Administration and Information* Technology, OTA-CIT-311 (Washington, DC: U.S. Government Printing Office, October 1986); *Hospital Information Systems at the Veterans' Administration,* OTA-CIT-372 (Washington, DC: U.S. Government Printing Office, October 1987).

4 U.S. Congress, Office of Technology Assessment, *The Social Security Administration and Information Technology,* op.cit., footnote 3.

its characteristics (see box 4-F). In 1988 an Association for Computing Machinery/IEEE Computer Society Task Force on the Core of Computer Science developed its detailed definition of the discipline through three paradigms: *theory* (rooted in mathematics), *abstraction or modeling* (rooted in the experimental scientific method), and *design* (rooted in engineering).

Government has a particular need, in future generations of software systems for well--engineered, maintainable software. An additional need is for tools and methods to plan for future software needs and ability to match technology to those needs in a timely manner. Several government projects aim at bringing government, industry, and academic research to bear on these projects. For example, a program called Software Technology for Adaptable, Reliable Systems works with industry to develop new software tools and methods. Part of this multiyear effort was the establishment of the Software Engineering Institute at Carnegie Mellon University which has done research on software reuse and other ''software factory' methods.

Technology Transfer

Software developed by the Federal Government may not be copyrighted. Under section 105 of the Copyright Act, copyright protection is not available for any work created by the Federal Government. Section 105 was enacted to give the public unlimited access to important information, to prevent the government from exercising censorship, and to prevent the government from using copyright in government works as a shield that would prevent selected groups from acquiring information.[5] In addition, it is argued that the public has paid for the creation of the work through taxes and should not pay a second time by paying copyright royalties.

Some propose that exceptions to the provisions of section 105 be made for computer programs, arguing that copyright protection for government software would facilitate its transfer to the private sector.[6] According to this view, private sector firms that might be interested in developing and marketing products based on government-developed software would be more likely to invest in the ' 'commercialization' of the government software if they were assured of an exclusive license.[7] Similar considerations have motivated government policy with respect to patents granted to the Federal Government. Opponents of an exception being made for computer programs argue that the exception is the ''thin end of the wedge," which could lead to further exceptions to section 105. In addition, it has been suggested that the line between programs and information' or 'data' is not always clear, and that granting exclusive rights to 'programs' could have the effect of limiting access to 'data' which would be retrieved using the programs.[8]

Legislation introduced in the 102d Congress would permit limited copyrighting of government software. H.R. 191 and S. 1581 would allow Federal agencies to secure copyright in software prepared by Federal employees in the context of cooperative research and development agreements (CRADAs) with industry.

---

5 Ralph Oman, Register of Copyrights, testimony at hearings before the House Subcommittee on Science, Research and Technology, Apr. 26, 1990, Serial No. 117, p. 100.

6 John M, Ols, Jr., Director i, the Resources, Community, and Economic Development Division, General Accounting Office, testimony at hearings before the House Subcommittee on Science, Research, and Technology, Apr. 26, 1990, Serial No. 117, p. 44.

7 Ibid., p. 41.

8 Steven J. Metalitz, Vice President and General Counsel, Information Industry Association, testimony at hearings before the Senate Committee on Commerce, Science and Transportation, Sept. 13, 1991.

SOURCE: OTA and cited sources.

Box 4-F—The Discipline of Computer Science

In March 1991, the Association for Computing Machinery (ACM) and the IEEE Computer Society (IEEE-CS) published a joint report on recommendations for undergraduate curricula in computer science. The report, ***Computing Curricula 1991,*** was prepared by the ACM/IEEE-CS Joint Curriculum Task Force and was intended to present "current thinking on goals and objectives for computing curricula. " The curriculum recommendations in the report built upon nine areas comprising the subject matter of the discipline:

1. algorithms and data structures,
2. architecture,
3. artificial intelligence and robotics,
4. database and information retrieval,
5. human-computer communication,
6. numerical and symbolic computation,
7. operating systems,
8. programmingg languages, and
9. software methodology and engineering,

In preparing this report, the task force drew upon the comprehensive definition of the discipline of computer science presented in 1988 by the ACM/IEEE-CS Task Force on the Core of Computer Science. In its 1988 report, ***Computing as a Discipline, the*** Task Force on the Core of Computer Science noted that it had extended its task to include computer engineering, as well as computer science because there was not fundamental difference between the core material for the two fields; the difference between them is that "computer science focuses on analysis and abstraction; computer engineering on abstraction and design. " The task force's definition of the "discipline of computing" included all of computer science and engineering:

> The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, "What can be (efficiently) automated?"

Concerning the role of programming languages, the Task Force on the Core of Computer Science had noted that the notion that "computer science equals programming" is misleading because many activities (such as hardware design, validating models, or designing a database application) are not "programming."Therefore, it concluded that computer science curricula should not be based on programming.Nonetheless, the task force did recommend that competence in programming be part of the curricula because:

> It is. . clear that access to the distinctions of any domain is given through language, and that most of the distinctions of computing are embodied in programming notations.

SOURCES: ACM/IEEE-CS Joint curriculum Task Force, *Computing Curricula 1991 (New York NY:* Association for Computing Machinery. 1991); "A Summary of the ACM/IEEE-CS Joint curriculum Task Force Report," Communications of the ACM, vo1. 34, No. 6, June 1991, pp. 69-84; and Peter J. Denning et al., "Computing as a Discipline,' *Communications of the ACM,* vol. 32, No. 1, January 1989, pp. 9-23.