18th BILETA Conference:*Controlling Information in the Online Environment*

April, 2003
QMW, London

From Idea to Action: Toward a Unified Theory of Software and the Law

Robert Plotkin
Concord, MA USA

Abstract

The application of various areas of law to software has proven troublesome. A general methodology is proposed for determining how a particular area of law should apply to software. The methodology asks and answers four questions: (1) "What is software?," (2) "How does software differ from other creative works?," (3) "Which of these differences, if any, are relevant to the law, and how?," and (4) "How should the law treat software in light of such differences?" Application of this methodology reveals that a computer program has the unique quality of being a set of human-readable instructions that describe actions in purely logical terms and which are capable of being executed automatically by a computer. This methodology is applied to two particular areas of law: intellectual property law and the First Amendment to the U.S. Constitution. In each case, software's unique features violate the law's assumptions, leading to results that are at odds with the policies underlying the law. Recommendations are made for modifying the legal rules in both areas to produce a better fit with software and which better promote the applicable underlying policies. Suggestions are made for future work in which the same methodology is applied to other areas of the law.

Introduction

Application of various areas of law to software has proven problematic and controversial. For example, the appropriate form of intellectual property protection for software is still being debated, [1] as is the question whether instructions written in computer source code are a form of expression protected by the First Amendment of the U.S. Constitution.[2] Although parties to such debates typically agree that there is something unique about software, often there is little agreement about the nature of software's distinctiveness or the relevance of software's uniqueness to its treatment by the law. [3]

This paper addresses the problem of how the law should apply to software by applying a methodology which answers four questions in sequence: (1) "What is software?," (2) "How does software differ from other creative works?," (3) "Which of these differences, if any, are relevant to the law, and how?," and (4) "How should the law treat software in light of such differences?" This methodology is applied to two particular areas of law (intellectual property law and first amendment law), and the potential generalizing the approach is explored.

I. What is software?

A computer program (also referred to as "software")[4] is a set of instructions that are capable of being executed[5] (performed) by a computer. For this reason, software is typically defined as "computer-executable instructions." A computer program is like a recipe that contains a sequence of instructions to be carried out by a computer.

A computer includes both hardware and software. The purpose of the hardware is to execute software. Computers include two essential hardware components: a processor and memory.[6] Computer programs may be stored in a computer's memory. The processor, sometimes referred to as the "brain" of the computer, executes ("runs") a computer program by reading individual program instructions from the memory and performing the instructions. Executing individual instructions may involve, for example, performing mathematical or logical operations, receiving input from the user or another device, providing output to a user or other device, or making decisions about which instructions to execute next.

Computers are sometimes referred to as "general-purpose computers" or "universal machines"[7] because computer hardware is like a blank slate that is capable of performing any function that may be specified by a computer program written in a programming language that the hardware understands. Computers are therefore "general-purpose" machines in the sense that they are capable of performing a near-infinite variety of functions, in contrast to "special-purpose" machines, such as hammers and pulleys, which are designed to perform a specific function. Although the hardware of a computer is generic and performs no useful function in the absence of software, each software program is specific in the sense that it specifies a particular function to be performed.

In this sense, computer hardware is to computer software as a drill is to a drill bit. The drill is generic in the sense that it does nothing useful in the absence of a drill bit, while a drill bit is specific in that it is designed to perform a specific function when attached to a drill. The act of storing a particular computer program in the memory of a general-purpose computer transforms the computer into a special-purpose machine for performing the function specified by the program,[8] just as connecting a drill bit to a drill transforms the drill into a special-purpose machine for performing the function specified by the drill bit. In light of these considerations, one may also view hardware as the fixed part of a computer and software as the variable part of a computer.

Although software is often described as "intangible" or "abstract,"[9] executable software stored in the memory of a computer is in fact a physical component of the computer. Although modern computers store software in the form of electromagnetic or optical signals which can neither be seen by the human eye nor manipulated by the human hand, such signals are "physical" in the sense that they are particular configurations of energy which physically interact with the computer's hardware to cause the computer to perform particular physical actions. Although it may appear counterintuitive to consider electrical signals to constitute a component of a machine, a machine is "an assemblage of parts that transmit forces, motion and energy one to another in a predetermined manner" or "an instrument . . . designed to transmit or modify the application of power, force, or motion."[10] With the advent of programmable computers, electrical signals are now capable of being configured into particular arrangements to perform all of these functions.[11]

Future developments in computer technology may produce software in physical forms that are more readily recognizable. For example, biological computers may embody software in DNA strands or other biological materials whose physical nature is more apparent than that of electrical signals.[12]

II. How does software differ from other creative works?

In this section I will argue that what distinguishes software from other classes of creative work – such as dramatic works and electromechanical machines – is the unique process by which software is created. The present analysis is limited to creative works whose intended end product is an action, whether it be the production of a play in the case of dramatic works, the movement of mechanical parts in the case of mechanical invention, the performance of a political action in the case of political

speechwriting, or the transmission of information over the Internet in the case of software. The unique nature of the software development process will be demonstrated by considering several kinds of creative works, tracing the creation of such works from the initial idea in the mind of the creator to the eventual resulting action, and comparing and contrasting the paths that lead from idea to action in each case.

A. The Life of a Software Idea

Consider the life of a computer program as it is transformed from an initial idea in the mind of a programmer into actions performed by a computer that executes the program. The following discussion uses an "email client" as an example of such a computer program. An email client is a computer program that sends and receives email through an email server.[13] Well-known examples of email clients include Microsoft® Outlook® and Qualcomm Eudora®. The basic functions performed by an email client include: (1) allowing the user to input (typically using a keyboard) an email message to send to another user; (2) transmitting email messages to other users; (3) receiving email messages transmitted by other users; and (4) displaying (typically on a computer monitor) email messages received from other users.

One way to model the creation of such a computer program is as a process including the following stages: (1) Problem Definition; (2) Requirements Analysis; (3) Logical Structural Design; (4) Construction; and (5) Execution.[14] This simplified model of the software development process is not intended to accurately reflect the way in which all software is actually designed, but rather to make clear certain features of the software development process which distinguish it from other creative processes that result in action.

In the problem definition stage of creating an email client program, the programmer may define the problem in terms of a high-level objective to be achieved,[15] e.g.: "to create a program for sending and receiving email messages over a network." In the requirements analysis stage, the programmer decides upon requirements that such a program must satisfy,[16] such as the ability to transmit messages including graphics or the ability to transmit messages to multiple recipients at once.

The logical structural design stage may be subdivided into several sub-stages: (a) architectural design; (b) detailed design; and (c) coding. Architectural design involves designing the high-level logical structure of the program, a task that is roughly analogous to writing a high-level outline for a book or legal brief.[17] The architecture of a software program specifies the high-level modules of the program in terms of the functions they perform and the interrelationships among them.[18] For example, the email client programmer may decide that the program should include four high-level modules, corresponding to the four high-level functions to be performed by the program: (1) an input module for receiving (typed) email messages from the user; (2) a transmission module for transmitting email messages to other users; (3) a reception module for receiving messages from other users; and (4) an output module for displaying email messages received from other users. Later in the software development process each module may be implemented as a distinct sub-program, although this is not required.

In the detailed design phase of the logical structural design stage, the programmer designs the program's modules (and their interrelationships) in more detail.[19] For example, the programmer may decide that the input module, which receives typed email messages from the user, is to include sub-modules for checking the spelling of typed email messages before they are sent and for looking up the email addresses of recipients who the user has specified by full name.

Note that until this point the programmer has not necessarily written any actual program code. This task is relegated to the coding phase of logical structural design, in which the programmer writes program code, in a programming language such as C or Java, which implements the detailed design that was created in the detailed design phase.[20] The result is a set of human-readable program instructions, referred to as "source code," which specifies the functions to be performed by the

program.

The next stage – construction – involves transforming the source code into an executable computer program in the form of electrical signals in the memory of the computer. Source code written in some programming languages must be "compiled" before it can be executed, which means that the source code must be translated into a "machine language" that is typically not readable by humans. Source code written in other programming languages may be executed directly by the computer. In either case, the construction stage is performed automatically by the computer in the sense that the computer automatically translates the source code into electrical signals having a particular physical structure designed to interact with the computer's hardware in a way that causes the hardware to carry out the functions specified by the source code. The automated nature of construction is more apparent in cases in which the source code must be compiled before it can be executed. Construction is still performed automatically by the computer, however, even in the case of non-compiled software in the sense that the act of typing the source code into the computer causes the computer to store appropriate electrical signals in the computer's memory without requiring the programmer to understand the physical structure or operation of the computer's hardware or the nature of the particular electrical signals that are created by the computer to implement the instructions in the source code.

In the final stage – execution – the computer automatically performs the instructions specified by the software. In the case of an email client, executing the program typically causes a window to be displayed in which a user may send and receive email and perform all of the other familiar functions performed by an email client.

B. Other Creative Processes

1. The Life of an Inventive Idea

Now consider the life of an idea for a conventional electromechanical[21] machine. The following discussion uses the telegraph, invented by Samuel Morse, as an example of such a machine.[22] As is well known, two telegraphs connected to each other over a long distance by a wire may transmit messages in Morse code over the wire in the form of electrical signals. Each telegraph includes a transmitter, receiver, means (e.g., a lever) for allowing a user to input messages in Morse code, and means for producing audible output representing received messages.

One way to model the creation of such an electromechanical machine is as a process including the following stages: (1) Problem Definition; (2) Requirements Analysis; (3) Logical Structural Design; (4) Physical Structural Design; and (5) Execution. The first two stages will not be described here because they do not differ in any relevant respects from the corresponding stages in the software development process described above in Section II.A.

One may imagine that Morse, in the logical structural design stage, decided that the telegraph should include four high-level modules (also referred to as "subsystems"): (1) an input module for receiving (tapped) messages from the user; (2) a transmission module for transmitting messages to other users; (3) a reception module for receiving messages from other users; and (4) an output module for outputting messages received from other users.

In the physical structural design stage, Morse designed particular physical structures for implementing the functional modules just described. For example, Morse designed a particular kind of signal lever, circuitry, receiver, and electromagnet, all arranged in a particular physical configuration and physically connected to each other in a particular way.[23] The resulting physical design, which may be depicted in a drawing or a written description, specified all of the physical features necessary to make and use a working telegraph.[24] Once the stage of physical structural design was complete, it became possible for Morse, or any skilled electrician/mechanic, to use ordinary skill to build a telegraph which could be used to transmit messages over a wire at distances

never before possible.

2. The Life of a Dramatic Idea

Having presented a general framework for conceptualizing creative processes, I will outline two other creative processes only briefly. Consider now the life of an idea for a dramatic work – a stage play in particular. One may imagine a playwright conceiving of an initial abstract idea for a theme, and then narrowing this idea and making it more and more concrete by engaging in the processes of plot development and scriptwriting. The script may then be provided to a producer, who (directly or indirectly) selects and organizes a theater company including a director, cast, and set designers, among others. The theater company interprets and rehearses the play, and then performs the play as rehearsed.

3. The Life of a Political Idea

Finally, consider the life of a political idea, by which I mean an idea for a political action to be performed. For purposes of example, consider an idea for a public rally in support of a particular political cause. Such a rally may begin as an idea in the mind of a politician, civic leader, or concerned citizen. The idea may be developed in the form of a written speech, which the speaker delivers to a potentially sympathetic audience in an attempt to encourage the audience to hold the rally. If the audience is persuaded by the speech, they may develop a plan to march to the town common and hold the rally, completing the chain from idea to action.

C. The Lives of Ideas Compared

Each of the creative processes described above may be divided into two high-level stages: design and implementation.[25] The term "design" refers herein to the process of refining an abstract idea into a more and more concrete idea that can be carried out automatically.[26] An action may be carried out automatically either by a human or by a machine, so long as it does not require the exercise of creativity. Completion of the design process requires the exercise of creativity, and therefore by definition design cannot be carried out automatically.

The term "implementation" refers herein to the process of embodying an idea in a physical form and/or action based on a completed design. Implementation can be performed automatically, and therefore by definition does not require the exercise of creativity to perform.[27]

Applying these concepts to the examples provided above, software design ends once the programmer has finished writing the program code (i.e., upon completion of the "coding" phase of the "logical structural design" stage), while implementation involves the automatic construction and execution of the program code by a computer. Conventional electromechanical design ends after the inventor (such as Morse) designs the physical structure of the machine, while implementation involves the construction and use of the machine. The design of a dramatic work ends after the final dress rehearsal, when all decisions about how to implement the play have been made and committed to memory, while implementation involves performing the play. The design of a political action ends after the plan to perform the action (e.g., a rally) is complete, while implementation involves the performance of the political action.

The exact location of the line dividing design and implementation in each case above is not important. Rather, what is important to recognize is that such a dividing line exists in each case, that such a dividing line separates the creative part of the process (design) from the non-creative part of the process (implementation), and that upon completion of design it becomes possible for the remainder of the process (implementation) to be performed automatically.

D. The Lives of Ideas Contrasted

The process by which software is created differs from the other creative processes described above in a critical way. The process of designing software ends upon the design of instructions expressed in purely logical terms. The instructions in a computer program are expressed in terms of mathematical and logical operations to be performed on abstract data, without specifying the physical hardware that is to be used to perform such operations, the physical structure of the software that is to be used to perform such operations, or the physical actions that are to be used to perform such operations. [28]

In all of the other creative processes described above, design does not end until after the conception of: (1) actions conceived in terms of physical movements; and/or (2) objects conceived in terms of their physical structure. For example, in the case of Morse's telegraph (an example of a conventional electromechanical machine), the design stage was not complete until Morse conceived of the physical structure of the telegraph. In the case of a play, design is not complete until the set and costumes are made and the cast and crew memorize the actions they are to perform during the performance. In the case of a political action, design is not complete until the political action itself is planned.

Once design is complete in each case, it is possible to implement the idea in a physical action or object without further exercise of creativity, but not before. Morse, for example, could not have constructed a working telegraph prior to designing the physical structure of the telegraph. Nor can a theater company perform a play before collectively designing and memorizing the movements of the players. Even improvisation requires the conception of physical actions to be performed, although in such a case at least some design and implementation occurs simultaneously. In no case is it possible to implement an idea in a physical form or action automatically based merely on the conception of instructions expressed in purely logical terms.

Computer programmers, however, complete the design stage upon the design of instructions expressed in purely logical terms. It is at this point that design ends and implementation begins. Upon the completion of such design, in other words, it becomes possible to implement the programmer's idea automatically, by providing the instructions (in the form of source code) to a computer and instructing the computer to execute the instructions.

The particular contrast between software design and conventional electromechanical design is illustrative. In conventional electromechanical design, it is necessary to design the physical structure of a machine in order to make and use the machine. Morse, for example, would not have been able to build a working telegraph if he had not conceived of the physical structure of the telegraph. Furthermore, it is necessary for an inventor to describe a new electromechanical machine in terms of its physical structure to enable others to make and use the machine. Morse, for example, would not have enabled others to make and use the telegraph merely by describing the telegraph as "a machine that includes an input subsystem, a receiver, a transmitter, and an output system suitable for sending and receiving long-distance messages over a wire using electricity."

Computer programmers, in contrast, need not design the physical structure of computer programs in order to make and use such programs or to enable others to do so. Rather, programmers need merely design and describe programs in terms of their logical structure by writing program instructions in a computer programming language. A computer automatically performs the act of transforming such instructions into a physical form suitable for performing the actions described by the instructions.

The process of software design ends and the process of implementation begins one stage earlier than in electromechanical design. Upon the completion of logical structural design, programmers turn over their source code to a computer, which performs the remainder of the process automatically, while electromechanical engineers must perform one more stage – physical structural design – manually by exercising creativity. Creative people in other fields must similarly continue to exercise creativity to transform instructions expressed in logical terms into physical actions or objects.

The software development process therefore represents a partial automation of the process linking idea to action. In particular, prior to the advent of computers the process of transforming instructions expressed in purely logical terms into a physical form and physical actions required the exercise of creativity and/or physical effort. This process can be performed automatically in the case of software, thereby freeing programmers (or any other human beings) from having to perform this transformation themselves. This is true whether one views software development as a process of inventing a machine component, an act of expressing ideas, or a combination of both.

III. How are software's unique qualities relevant to the law?

A. Overview

The law in various areas makes assumptions about where (creative) design ends and (rote) implementation begins. In other words, the law makes assumptions about where in the creative process it becomes possible to implement an idea automatically. Prior to the advent of computers, these assumptions were generally valid and therefore typically produced the correct results when applied in particular cases.

Software, however, violates the law's assumptions about where design ends and implementation begins. As a result, applying the old legal rules to software produces the wrong results. In this section I provide justification for these claims using two areas of law as examples: intellectual property law[29] and first amendment law.

B. Case Study 1: Patent Law

Patent law embodies certain assumptions about the inventive process.[30] For example, I assert that patent law assumes that, as a general rule: (1) machines and components thereof are conceived of in terms of their physical structure; (2)(a) once a machine has been designed, it is possible to describe the machine in terms of its physical structure; (2)(b) once a process (as that term is defined within patent law) has been designed, it is possible to describe the process in terms of its constituent physical actions; (3)(a) it is necessary to describe a machine in terms of its physical structure to enable others to make and use it; and (3)(b) it is necessary to describe a process in terms of physical actions to enable others to perform the process.

These assumptions are reflected in patent law's rules of patentability (which determine whether a purported invention is patentable) and claim scope (which determine the scope (breadth) of protection to be given to a particular invention). Whether a machine is patentable is determined by reference to the physical structure of the machine. For example, as a general rule a product invention is not considered complete until the inventor has conceived of the invention in terms of its physical structure.[31] Similarly, a machine satisfies patent law's novelty requirement if and only if the machine's physical structure differs from the physical structure of previous machines (referred to as the "prior art").[32] A corollary to this rule is that failure by the patent applicant to describe the physical structure of a machine for which a patent is sought will result in denial of a patent for the machine.[33] Similarly, failure by the patent applicant to conceive of and describe particular physical means for carrying out a process for which a patent is sought will likely result in denial of a patent for the process, although the law on processes is less clear.[34]

Furthermore, the scope of legal protection granted to a particular invention is ascertained by reference to the physical structure of the invention (whether machine or process), as disclosed and claimed by the inventor in the patent.[35] These rules, in combination, impose what are referred to as "physicality" requirements on machines and processes for which patents are sought. "Substantive" physicality requirements (such as novelty and nonobviousness) require that the invention itself be capable of being embodied in a physical form, while "formal" physicality requirements (such as enablement, written description, and particularized claiming) require that the patent applicant describe and claim the invention in terms of physical objects or actions in the patent application.[36]

Software is inconsistent with all of the above-stated assumptions of patent law. In particular: (1) software programs are physical machine components that are not conceived of in terms of their physical structure; (2) software programs typically cannot be described in terms of their physical structure; and (3) it is not necessary to describe a software program in terms of its physical structure to enable others to make and use it. Brief justifications for each of these assertions are that: (1) software programs are conceived in terms of their logical structure, not their physical structure; (2) software programs are embodied in the form of electrical signals which cannot readily be conceptualized or described in terms of their physical structure; and (3) to enable a software program to be constructed and used it is necessary only to describe the program in terms of its logical structure (e.g., using source code).[37]

The incompatibility of software with the above-stated assumptions of patent law causes patent law to produce the wrong results, in at least some cases, when applied to software. Most generally, these wrong results include both underprotection of software and overprotection of software. Software is underprotected when patent protection is denied to a program that is worthy of patent protection because it is useful, novel, and nonobvious, or when the scope of protection granted to a program is narrower than what the inventor has enabled the public to make and use. Underprotection may result from the inconsistencies stated above when the benign and unavoidable inability of a programmer to describe a program in terms of its physical structure or the physical actions it performs causes patent protection to be denied to the program, despite the fact that the program is useful, novel, and nonobvious.[38]

Software is overprotected when patent protection is granted to a program that does not satisfy all of the requirements for patentability, or when the scope of protection granted to the program is broader than what the inventor has enabled the public to make and use. Overprotection may result from the inconsistencies stated above when a lack of physical terms in the specification and/or claims of a software patent causes the patent's claims to be interpreted broadly to encompass any machine or process for performing the same function as the patented software, even though the patent's specification only enables the public to make and use a limited range of software for performing such a function.[39]

C. Case Study 2: First Amendment law:

First amendment law also embodies certain assumptions about how instructions may be transformed into action. For example, I assert that first amendment law assumes that, as a general rule, instructions require at least some human agency to carry out. As a result, first amendment law also assumes that the mere expression of instructions (whether verbally or in writing) typically is insufficient to enable those instructions to be carried out automatically. Cases in which the mere expression of instructions can cause the instructions to be carried out essentially automatically, as in cases of duress, are considered the exception rather than the rule.
These assumptions are reflected in the legal rules that have developed to determine whether a particular speaker may be held liable for harm caused by his speech. Such rules include rules that: (1) mere expression of instructions typically is protected speech;[40] and (2) expression may be unprotected or otherwise actionable if the speaker has sufficient intent to cause harm and the causal chain linking the expression to harm is sufficiently strong.[41] Intent to cause harm may be inferred from a close causal connection between speech and resulting harm.[42] The present discussion focuses in particular on cases in which a speaker expresses instructions to a listener who carries out the instructions and thereby causes harm to a third party, as in the case of a fiery speaker who incites an angry mob to assault an innocent bystander.

Software is inconsistent with the above-stated assumptions of first amendment law. In particular, instructions expressed in software do not require any human agency to carry out. As a result, the mere expression of instructions in the form of software can be sufficient to enable those instructions to be carried out automatically. Furthermore, in the case of software, a close causal connection

between speech embodied in software and harm resulting from that speech does not necessarily imply that the speaker-programmer intended to cause such harm. A computer science professor, for example, who publishes the source code for a virus in a textbook on computer security likely does not intend for such source code to cause harm, even if there is a close causal connection between publication of the textbook and the malicious redistribution of the virus code in executable form by a third party.

The incompatibility of software with the above-stated assumptions of first amendment law may cause first amendment law to produce the wrong results, in at least some cases, when applied to software. Most generally, these results include both underprotection of "software speech" (speech embodied in software code) and overprotection of software speech. Software speech is underprotected in cases in which the speaker-programmer is held liable for software that causes harm or is prohibited from distributing software out of concern for harm that it may cause, even though the speaker-programmer: (1) distributed the software solely or primarily for expressive purposes; and (2) took reasonable precautions to prevent harm from resulting.[43]

Underprotection may result from the inconsistencies stated above when the law fails to recognize that software distributed with expressive intent, benign motives, and reasonable care may be misappropriated by others to cause harm relatively easily due to software's ability to be executed automatically and distributed electronically at essentially no cost.

Software speech is overprotected in cases in which the speaker-programmer is not held liable for software that actually causes harm, even though the speaker-programmer: (1) knew or should have known that distribution of the software would cause harm; and (2) did not take reasonable precautions to prevent harm from resulting. Overprotection may result from the inconsistencies stated above when the law fails to identify cases in which a particular programmer-speaker should have taken the automatic executability of software into account when evaluating the likelihood that distributing such software would cause harm and when determining which precautions, if any, to take when distributing such software.

IV. Recommendations: methodology

Having identified ways in which intellectual property law and first amendment law may produce incorrect results when applied to software, I recommend changes to the legal rules in these areas of law which would produce results that better promote the policies underlying each area of law. More generally, I provide a methodology for modifying these and other areas of law to make them more consistent with the unique manner in which software is created and used. Applying this methodology to additional areas of law could lead to a more unified approach to the emerging field of "software law."

The methodology I propose includes two high-level steps: (1) develop a clear conception of software and the manner in which it is created; and (2) modify the relevant legal rules based on the conception of software developed in step (1). Step (2) is to be applied for each area of law, and may be further divided into three sub-steps: (a) identify (possibly unstated) premises underlying the relevant area of law that are inconsistent with the conception of software developed in step (1); (b) modify the identified premises to be consistent with the conception of software developed in step (1); and (c) update the relevant legal rules based on the modified premises.

I put forth a solution to step (1) – developing a conception of software – in Sections II and III, above. I put forth a solution to step (2)(a) – identifying premises that are inconsistent with the nature of software – in Sections IV.B and IV.C, above, in the cases of patent law and first amendment law. I will now propose solutions to steps (2)(b) and (2)(c) – modifying the premises and rules of the law – in the cases of patent law and first amendment law.

A. Recommendations: patent law

Patent law should adopt the following premises: (1) it is possible to build a (physical) executable software program by conceiving of the program solely in terms of its logical structure; (2) it is possible to enable others to make and use a (physical) executable software program by describing the program solely in terms of its logical structure; and (3) the scope of what a programmer has enabled others to make and use may be ascertained by reference to a description of the program expressed solely in terms of its logical structure. Among the conclusions that follow from these premises are that: (1) the lack of physical terms in a description[44] of a software program is not evidence that the programmer has failed to enable a physical executable software program to be made and used; and (2) the lack of physical terms in a description of a software program does not imply that there is no basis for ascertaining the scope of what the programmer has invented.

Based on these premises, I recommend that the rules of patentability and claim scope should be updated to apply to the logical structure in a manner that is analogous to the way in which such rules currently apply to the physical structure of electromechanical inventions. For example, the "statutory subject matter" rule, which requires that a purported invention be a machine, article of manufacture, composition of matter, or process, should require that a software program having the logical structure claimed in a patent application be implementable in a tangible embodiment (such as an executable software program) by those of ordinary skill in the art without undue experimentation. The tests for novelty and obviousness should compare the logical structure claimed by the patent applicant to prior art logical structures. The written description and enablement requirements should require that the patent applicant describe the logical structure of the claimed program and enable the public to make and use a program having the claimed logical structure. Finally, the scope of a software patent claim should be defined by the logical structure of the claimed program.

Such modifications to patent law would be premised on a more accurate understanding of the nature of software, the process by which it is invented, and the way in which programmers naturally think about and describe software. Furthermore, patent law is the best starting point from which to make such modifications because patent law already includes a variety of mechanisms, which would be useful in any intellectual property system that applies to software. These include mechanisms for: (1) determining whether a design is new, useful, and nonobvious; (2) distinguishing among embodiments of a design, technical descriptions of such embodiments, and descriptions of the novel and nonobvious features of the design; (3) providing a scope of protection that is tailored to the scope of invention in each case; and (4) notifying the public of the scope of protection in each case.

B. Recommendations: First Amendment law

First amendment law should adopt the following premises: (1) instructions written in software code are capable of being carried out automatically; and (2) the fact that software instructions have been used to cause harm in a particular case does not imply that the programmer intended the harm to occur, even if there is a close causal connection between the programmer's distribution of the software and the resulting harm.

There is precedent for using the First Amendment to modify the liability rules that are embodied in the elements of a criminal or civil cause of action. The holdings in the *Sullivan*[45] and *Brandenberg* [46] cases cited above, for example, have been use to impose heightened intent and proximate cause requirements in the context of various criminal and civil causes of action. *Sullivan*, in particular, applies in cases involving allegedly defamatory statements made about public figures. Although the class of cases in which the *Brandenberg* standard applies is difficult to define, it at least includes cases involving speech that is alleged to have actually incited unlawful action or have a tendency to incite such action.

Based on the premises set forth above, I recommend that heightened intent and proximate cause requirements should be applied in cases involving protected software speech, where "protected" speech is defined as that class of speech that the First Amendment was intended to protect, such as

political, religious, academic, and artistic speech. For example, heightened intent and proximate cause requirements should be applied in a suit for damages against a computer science professor who publishes virus source code in an academic journal for educational purposes and with benign motives, and in which a malicious third party redistributes the source code in a deceptive email message, thereby propagating the virus and causing widespread damage to electronic data. Applying appropriate heightened intent and causation requirements in such cases would provide sufficient protection for socially valuable speech while enabling liability to be imposed in cases in which speech has a sufficiently close causal connection to harm and in which the speech is published with a sufficient degree of intent to cause harm.

C. Conclusions

Software source code is unique among human creations in that it is a set of instructions written in a language that is both comprehensible by humans and capable of being executed automatically by computers. This distinctive combination of features upsets assumptions made by various areas of law about the properties of instructions and, more generally, about the manner in which ideas can be transformed into actions. As a result, the application of old legal rules to software has produced results that are at best surprising and at worst contrary to the policies underlying the law. The legal profession has struggled to grapple with this problem without developing a clear conception of the nature of software and without applying a methodology that can be applied clearly and consistently across multiple areas of law. This paper attempts to address this meta-problem by proposing a generally-applicable methodology and applying it to two particular areas of law – intellectual property law and first amendment law – which have proven particularly controversial in their application to software. Future work will attempt to apply the same methodology to other areas of law both to determine whether the methodology is truly generalizable and to solve the particular problems that software continues to pose as it runs up against an increasingly broad range of ancient legal rules developed without consideration of software's unique properties.

---

[1] The debate over software patentability in the U.S., for example, dates back at least to 1965, when President Johnson commissioned a comprehensive study of the United States patent system. The Commission explored a wide range of pressing issues facing the patent system and recommended, among other things, that "no patents on . . . computer programs" be issued. 1996 Report of the President's Comm'n on the Patent Sys. 1. The U.S. Supreme Court first wrestled with software patentability in Gottschalk v. Benson, 409 U.S. 63 (1972). For discussions of software patentability, *see generally* Vincent Chiappetta, *Patentability of Computer Software Instruction as an "Article of Manufacture:" Software as Such as the Right Stuff*, 17 J. MARSHALL J. COMPUTER & INFO. L 89, 143 (1998); Allen B. Wagner, *Patenting Computer Science: Are Computer Instruction Writings Patentable?,* 17 J. MARSHALL J. COMPUTER & INFO. L. 5 (1998); Christopher S. Cantzler, *State Street: Leading the Way to Consistency for Patentability of Computer Software*, 71 U. COLO. L. REV. 423 (2000); Julie E. Cohen & Mark A. Lemley, *Patent Scope And Innovation In The Software Industry*, 89 Cal. L. Rev. 1 (2001). For discussion of software copyrightability, *see generally* Marla R. Bloch, *The Expansion Of The Berne Convention And The Universal Copyright Convention To Protect Computer Software And Future Intellectual Property,* 11 Brook. J. Int'l L. 283 (1985); Michael A. Dryja, *Looking To The Changing Nature Of Software For Clues To Its Protection,* 3 U. Balt. Intell. Prop. L.J. 109 (1995).

[2] For example, federal regulations limiting the export of encryption technology have been challenged on the ground that such regulations violate the First Amendment rights of cryptographic

professionals to publish the source code to encryption software as part of professional and academic discourse on cryptographic techniques. *See, e.g.,* Bernstein v. U.S. Dep't of Justice, 176 F.3d 1132 (9th Cir. 1999); Junger v. Daley, 209 F. 3d 481 at 484-485 (6th Cir. 2000); Karn v. United States Department of State 925 F.Supp. 1 (D.C. Cir. 1996). For more information about the encryption cases, s*ee generally* Norman Andrew Crain, *Bernstein, Karn, and Junger: Constitutional Challenges to Cryptographic Regulations*, 50 Ala. L. Rev. 869 (1999).
More recently, the Digital Millennium Copyright Act (DMCA) has been challenged on the ground that its prohibition against trafficking in devices that circumvent certain copy-protection and access-control measures violates the First Amendment insofar as it prohibits the distribution of software code for performing such functions. *See, e.g.,* U.S. v. Elcom Ltd., 2002 U.S. Dist. Lexis 9161 (N.D. Cal. May 8, 2002); Universal City Studios, Inc. v. Corley, 273 F.3d 429; DVD Copy Control Ass'n v. Bunner, 113 Cal.Rptr.2d 338 (2001).
[3] For examples of arguments about the relevance of software's unique qualities to the law, s*ee, e.g.*, James R. Goodman et al., *Toward a Fact-Based Standard for Determining Whether Programmed Computers are Patentable Subject Matter: The Scientific Wisdom of* Alappat *and Ignorance of* Trovato, 77 JPTOS 353 (1995) (the particular manner in which computers are programmed using software distinguishes software from other technologies in a way that is relevant to patent law); Randall Davis et al., *Symposium: Toward a Third Intellectual Property Paradigm: A Manifesto Concerning the Legal Protection of Computer Programs*, 94 COLUM. L. REV. 2308, 2327 (1994) (computer programs are "machines . . . that happen to have been constructed in the medium of text" and therefore differ from other kinds of machines and other kinds of textual works for purposes of intellectual property protection).
[4] The terms "software" and "computer programs" are used interchangeably herein. For a dictionary definition of "software," *see* MICROSOFT COMPUTER DICTIONARY 489 (5th ed. 2002) (defining "software" as "[c]omputer programs; instructions that make hardware work").
[5] The term "executable" means "[c]apable of being run on [a] computer." WEBSTER'S NEW WORLD COMPUTER DICTIONARY 341 (9th ed. 2001).
[6] *See, e.g.,* GEORGES IFRAH, THE UNIVERSAL HISTORY OF COMPUTING: FROM THE ABACUS TO THE QUANTUM COMPUTER 313 (2001).
[7] Modern digital computers are also referred to alternatively as "universal Turing machines" and, by abbreviation, as "Turing machines" and "universal machines." The "universal machine" is also referred to as the "universal Turing machine," which is capable of mimicking any of a near-infinite number of special-purpose machines referred to as "Turing machines."
[8] *See In re* Alappat, 33 F.3d 1526, 1545 (Fed. Cir. 1994); *In re* Bernhart, 417 F.2d 1395, 1400 (C.C.P.A. 1969).
[9] *See, e.g.* James Gleick, *Patently Absurd*, N.Y. TIMES MAGAZINE, Mar. 12, 2000, at 44; *Prepared Testimony and Statement for the Record of Jim Warren Before the Patent and Trademark Office* (Jan. 26-27, 1994), *available at* http://www.bustpatents.com/autodesk.htm; John Perry Barlow, *The Economy of Ideas*, WIRED (Mar. 1994).
[10] *See* MERRIAM WEBSTER'S COLLEGIATE DICTIONARY 697 (10th ed. 1993).
[11] *See generally* STEPHEN A. WARD & ROBERT H. HALSTEAD, JR., COMPUTATION STRUCTURES (1990), especially pages 281-512 for an introduction to the interaction between stored electrical software and computer hardware.
[12] *See, e.g.,* Thomas F. Knight, Jr. & Gerald Jay Sussman, *Cellular Gate Technology*, MIT ARTIFICIAL INTELLIGENCE LABORATORY, (1998); Leonard Adleman, *Molecular Computation of Solutions to Combinatorial Problems*, 266 SCIENCE 1021-23 (1994); Dan Boneh et. al., *On the Computational Power of DNA*, 71 DISCRETE APPLIED MATHEMATICS, SPECIAL ISSUE ON COMPUTATIONAL MOLECULAR BIOLOGY, 79-94 (1996); Sandeep Junnarkar, *In Just a Few Drops, a Breakthrough in Computing*, N.Y. TIMES, May 21, 1997.
[13] For a definition of "client" and "server," *see* MICROSOFT COMPUTER DICTIONARY 102 (5th ed. 2002).
[14] For discussions of various models of engineering design, *see generally* GERARD VOLAND, ENGINEERING BY DESIGN (1999); NAM P. SUH, THE PRINCIPLES OF DESIGN (1990); ATILA ERTAS & JESSE C. JONES, THE ENGINEERING DESIGN PROCESS (2d ed. 1993).
[15] *See, e.g., generally* JAMES F. PETERS & WITOLD PEDRYCZ, SOFTWARE

ENGINEERING: AN ENGINEERING APPROACH 119-20 (2000); SUH, *supra* note 15, at 30-35.
[16] *See, e.g.,* PETERS & PEDRYCZ, *supra* note 15, at 117-122; The Engineering Design Process at 12-13.
[17] *See, e.g.,* WEBSTER'S NEW WORLD COMPUTER DICTIONARY 24 (9th ed. 2001) (defining "architecture" as "[t]he overall conceptual design and design philosophy of a hardware device or computer system or network"). In the context of software development, "[i]t has become common practice to use the term *architecture* to characterize the internal structure of a software system . . ." PETERS & PEDRYCZ, *supra* note 15, at 206. *See also* HAFEDH MILI ET AL., REUSE-BASED SOFTWARE ENGINEERING: TECHNIQUES, ORGANIZATION, AND CONTROLS 382-393 (John Wiley & Sons, Inc. 2002).
[18] *See, e.g.,* DICK HAMLET AND JOE MAYBEE, THE ENGINEERING OF SOFTWARE: TECHNICAL FOUNDATIONS FOR THE INDIVIDUAL 89-90 (2001).
[19] For an extensive treatment of detailed design, *see generally* STEVE MCCONNELL, CODE COMPLETE: A PRACTICAL HANDBOOK OF SOFTWARE CONSTRUCTION (1993).
[20] The phase of Logical Structural Design that I refer to as "coding" is often referred to in the literature, confusingly enough, as "construction." The conventional meaning of the "construction" in the context of computer programming refers to the process of "constructing" the *logical* structure of a program by writing source code, rather than to the process of constructing the *physical* structure of the program. For example, the term "construction" in the titles of the books *Code Complete: A Practical Handbook of Software Construction* and *Object-Oriented Software Construction* refer to the detailed design of computer source code, not to the kind of physical construction associated with conventional electromechanical devices. *See* MCCONNELL, *supra* note 20, at 1-6; *see also* BERTRAND MEYER, OBJECT-ORIENTED SOFTWARE CONSTRUCTION (2d ed. 1997). This usage of the term "construction" is correct by analogy in that it refers to the final and most detailed stage carried out by a human being in both electromechanical and software design. The use of the term "construction" to refer to the design of computer source code, however, may be confusing in the context of the present discussion. I therefore only use the term "construction" to refer to the physical construction of a physical entity (such as an executable software program), not to any aspect of the process of designing a software program or other product.
[21] The term "electromechanical" is defined herein as "electrical, mechanical, or a combination of both."
[22] I have used facts described in the Supreme Court case of O'Reilly v. Morse, 56 U.S. 62 (1853), whenever possible. In cases where relevant facts are not available in O'Reilly v. Morse, I use hypothetical facts and indicate as such in the text.
[23] In the case of the telegraph, Physical Structural Design involved selecting and/or designing the particular physical components of the telegraph (e.g., lever and circuitry) and the physical interconnections among them.
[24] The term "design" is ambiguous in that it refers both to the *process* of designing a product and to the resulting *design* for the product. *See* SHARI LAWRENCE PFLEEGER, SOFTWARE ENGINEERING: THEORY AND PRACTICE 195 (2nd Ed. 2001) ("Design is the creative process of transforming the problem into a solution; the description of a solution is also called [a] design."). The term "design" as used herein refers to an ideal entity (in the Platonic sense) that specifies the physical structure of a product. A design for a physical product may be tangibly embodied in a *physical design specification*, such as a schematic drawing of a telegraph. An architectural blueprint for a house is another example of a physical design specification.
[25] The terms "design" and "implementation" are used herein with meanings that closely track the meanings of the patent law terms "conception" and "reduction to practice." In patent law, conception has been said to be the "touchstone of inventorship, the completion of the mental part of invention." Burroughs Wellcome Co. v. Barr Lab., 40 F.3d at 1227-28. Patent law recognizes two kinds of reduction to practice: constructive and actual. The filing of an application for a patent disclosing the invention in the appropriate manner constitutes constructive reduction to practice. *See, e.g.,* Pfaff v. Wells Electronics Inc., 525 U.S. 55, 61 (1998). Actual reduction to practice occurs when the inventor (1) constructs a product or performs a process that is within the scope of the patent claims, and (2) demonstrates the capacity of the inventive idea to achieve its intended result. Chisum § 10.06; Eaton v. Evants, 204 F.3d 1094 (Fed. Cir. 2000); Genentech Inc. v. Chiron Corp., 220 F.3d 1345 (Fed. Cir.

2000).

[26] Providing a rigorous definition of automation is beyond the scope of this article. At least two plausible meanings, however, are sufficient for purposes of the present discussion. In one sense, an action is "automatic" if its execution does not involve the performance of substantial *physical* acts by a human being. In another sense, the term "automatic" refers to an action that is performed without all but the most trivial *mental* involvement by a human being. In particular, the term "automatic" in this sense refers to an action that is performed without the substantial exercise of conscious human thought or discretion.

[27] In patent law, once the inventor has conceived of the invention, "[a]ll at remains to be accomplished in order to perfect the act or instrument belongs to the department of construction, not invention." Mergenthaler v. Scudder, 11 App. D.C. 264, 1897 C.D. 724 (D.C. Cir. 1897). As a result, "[t]he true date of invention is at the point where the work of the inventor ceases and the work of the mechanic begins." Cameron & Everett v. Brick, 1871 C.D. 89 (Comm'r Pat. 187l).

[28] Consider, for example, the instruction "Add(2,4)," which is an instruction to add the numbers two and four. This instruction does not specify any particular hardware for performing the instruction, the physical structure of the software that is to be used to perform the instruction, or the physical actions that the computer is to use to perform the instruction.

[29] Although controversy has surrounded the application of both copyright and patent law to software, I focus in this paper on patent law because it is designed to protect machines and processes which perform useful functions, i.e., machines and processes that perform actions. Copyright law, in contrast, is designed to protect the expression of ideas. Patent law is therefore a better vehicle than copyright law for exploring the way in which intellectual property law interacts with the path from idea to action.

[30] Patent law expressly disavows any requirement that an invention be invented in a particular way to merit patent protection. *See, e.g.,* 35 U.S.C. § 103(a) ("Patentability shall not be negatived by the manner in which the invention was made."). Despite this, the rules of patentability and claim scope reflect implicit assumptions about requirements of real-world inventive processes.

[31] The Federal Circuit has held, in at least some cases, that an inventor of a product invention must form a "mental picture" of the invention's physical structure to satisfy the conception requirement. *See* Burroughs Wellcome Co. v. Barr Lab., 40 F.3d 1223, 1228 (Fed. Cir. 1994); Fiers v. Revel, 984 F.2d 1164, 1169 (Fed. Cir. 1993); Amgen, Inc. v. Chugai Pharmaceutical Co., 927 F.2d 1200, 1206 (Fed. Cir. 1991).

[32] A product claim covers what a product *is*, not what the product *does*. Hewlett-Packard Company v. Bausch & Lomb Incorporated, 909 F.2d 1464 (Fed. Cir. 1990). In the case of a product claim that defines the product in terms of its physical structure, the claim only satisfies the novelty requirement if no single prior art source discloses all of the physical structure recited in the claim. *See, e.g.,* Del Mar Eng'g Laboratories v. Physio-Tronics, Inc., 642 F.2d 1167, 1172 (9th Cir. 1981) ("To overcome the defense of anticipation, it is only necessary for the patentee to show some tangible difference between the invention and the prior art."); Schroeder v. Owens-Corning Fiberglas Corp., 514 F.2d 901, 185 U.S.P.Q. (BNA) 723 (9th Cir. 1975) ("An invention is said to be anticipated only if another invention already known or used is identical in substance."). In the case of a product claim that defines the product in terms of the functions it performs, the claim is limited in scope to the particular physical structures disclosed in the patent specification. 35 U.S.C. § 112. Therefore, the novelty of a claimed machine is determined by reference to the machine's physical structure, whether the machine is claimed in terms of its physical structure or the functions it performs.

[33] *In re* Donaldson Co., 16 F.3d 1189, 1195 (Fed. Cir. 1994) (en banc). In *In re* Wilder, 736 F.2d 1516, 1521 (Fed. Cir. 1984), however, the Court determined that a purely functional description may be sufficient if the function described is known to correlate to a specific structure. *See* "Guidelines for the Examination of Patent Applications Under the 35 U.S.C. § 112, ¶ 1 'Written Description' Requirement," § II.A.1.3.a, USPTO, Revised 2001.

[34] *See* Alpert v. Slatin, 49 C.C.P.A. 1343, 1347 (C.C.P.A. 1962) ("Conception of an inventive process involves proof of mental possession of the steps of an operative process and, if necessary, of means to carry it out to such a degree that nothing remains but routine skill for effectuation thereof."); Rey-Bellet v. Engelhardt, 493 F.2d 1380 (C.C.P.A. 1974); Amax Fly Ash Corp. v. United

States, 206 Ct. Cl. 756, 770 (1975).
[35] Robert Plotkin, "Computer Programming and the Automation of Invention: A Case for Software Patent Reform," unpublished manuscript on file with author.
[36] In particular, the Federal Circuit has interpreted the written description requirement to require that, as a general rule, inventors describe product inventions in terms of their physical features. *See* Fiers, 984 F.2d at 1170-71; *accord* Eli Lilly & Co., 119 F.3d at 1559. Although essentially all written description cases have involved product claims, it appears that the written description requirement is satisfied for a process claim by describing the particular steps that comprise the process. *See In re* Kaslow, 707 F.2d 1366 (Fed. Cir. 1983).
[37] See Section II, supra.
[38] For example, in one case the Federal Circuit initially denied patent protection on the grounds that "the specifications involved here provide no grasp of any underlying physical processes" and "no computer architecture is provided, no circuit diagram is revealed, and no hardware at all receives more than a brief mention." *In re Trovato*, 42 F.3d 1376 (Fed. Cir. 1994), *vacated and withdrawn by* 60 F.3d 807 (Fed. Cir. 1995).
[39] Concern about overbreadth in this sense was the Supreme Court's primary reason for invalidating claim 8 of Morse's telegraph patent. O'Reilly 56 U.S. at 113. Software patents that describe and claim software in purely functional terms are susceptible to being overbroad in the same way as Morse's claim 8. Critiques of software patents that are based on the claim that software is "abstract" or "intangible" may be interpreted to mean that *descriptions* of software tend to be abstract in the sense that such descriptions describe software purely in purely logical terms, and are therefore susceptible to broad interpretations.
[40] *See, e.g.,* Brandenburg v. Ohio, 395 U.S. 444 at 447 (1969) (holding that mere "advocacy of the use of force or of law violation" is not actionable "except where such advocacy is directed to inciting or producing imminent lawless action and is likely to incite or produce such action.").
[41] For example, in the case of allegedly defamatory statements made against a public figure, the First Amendment requires a heightened degree of intent (referred to as "actual malice") in comparison to the degree of intent that is normally required in a defamation claim. NY Times v. Sullivan, 376 U.S. 254 (1964). In the case of speech that allegedly constitutes an incitement to unlawful action, the First Amendment requires that such speech be "directed to inciting or producing imminent unlawful action and is likely to incite or produce such action." Brandenburg v. Ohio, 395 U.S. 444 at 447 (1969). This standard imposes both a heightened intent requirement and a heightened causation requirement.
[42] Planned Parenthood of the Colom./Willamette, Inc. v. Am. Coalition of Life Activists, 2002 U.S. App. Lexis 9314 (9th Cir. 2002).
[43] The case of Princeton University professor Edward Felten would have been such a case had his case gone to trial and resulted in a finding of liability against him. *See* "Security Researchers Drop Scientific Censorship Case," Electronic Frontier Foundation Media Release, February 6, 2002, available at
http://www.eff.org/IP/DMCA/Felten_v_RIAA/20020206_eff_felten_pr.html.
[44] Although in patent law the term "description" typically refers to the specification of the patent, the term "description" in this paragraph encompasses both the specification and claims of a patent, as well as other descriptions of a software program.
[45] Sullivan, 376 U.S. at 254.
[46] Brandenburg, 395 U.S. at 444.