Report

A New View of Intellectual Property and Software

Randall Davis

Pamela Samuelson

Mitchell Kapor

Jerome Reichman

Each time there appears to be a lull in the controversy about legal protection for software, we are quickly jolted by the battle being joined anew. The difficulties won't soon disappear, we believe, because there is a deep-seated problem here: existing intellectual property laws are fundamentally ill-suited to software. The problems are rooted in the core assumptions in the law and their mismatch with what we take to be important about software.

We believe a durable solution requires a new approach to the problem. We suggest one such approach that is founded on the notion of *market preservation*, that is, constructing just enough machinery to head off the ways in which marketplaces fail. We outline here how to do this using a law focused on short-term protection against clones and that solves the market failure problem without interfering unduly with the ability to reuse incremental innovation. A more extensive discussion of our ideas is found in [3]. This overview focuses on the technical issues of particular interest to this audience.

Preliminaries

Two brief points will help establish context. First, in talking about the software marketplace, we are focusing primarily on mass market software, which basically means personal computing software (though we believe much of what we offer carries over to other software markets). Where product volume is small enough (e.g., mainframe software) to permit it, individual sales contracts can be negotiated to meet the individual needs of buyer and seller. When products are sold in the tens of thousands to millions of units, only standard, agreed-on rules are feasible, rules of the sort provided by intellectual property law.

Second, some of what we say will seem obvious to a technical audience and perhaps even seem anticlimactic. This happens because our effort over the past few years has been in part one of debugging, uncovering the unarticulated assumptions and unspoken misconceptions in both the legal and technical communities that lay at the source of the problem. As in many debugging efforts, understanding the source of the difficulty was the hard part; the steps needed for repair may by comparison be relatively simple. Nevertheless, we argue with some vigor that while the changes may be small, they are necessary; the existing system cannot work in the long term.

Our industry lacks an agreed-on set of rules for competition. What are, or ought to be, the rules of fair following?

What's Wrong?

There is considerable and fundamental controversy over intellectual property laws for software. In this area (and in a variety of other high-technology areas) the stakes are significant and the controversies are basic: What aspects of a program are copyrightable? How much more than the literal work is covered? What elements of software can be covered by a patent? Cases routinely arise in applications of intellectual property law to traditional subject matter such as novels, movies, and so forth, but there the issue is typically one of fine tuning the rules at the margins, not making the sort of fundamental decisions that routinely arise in software cases.

Our industry lacks an agreed-on set of rules for competition. What are, or ought to be, the rules of fair following?

The rules are not only unclear, they keep changing. In copyright, for example, the past decade has seen a sequence of cases that have made major differences in the interpretation of the law.[1]

Supporters of the existing legal system suggest the solution is simply to wait (translation: the uncertainty is a transient that will soon damp out). They point out, correctly, that software law is relatively young and that the U.S. legal system has successfully been adapting to technological innovation for more than 200 years (photographs, motion pictures, audio and video recorders). This argument is credible and needs to be taken seriously. However, we believe it is incorrect.

We suggest the problem is more fundamental: existing legal concepts frame the issues badly when it comes to software, and additional experience will only produce more thrashing.

As one example of the conceptual difficulty, consider that, according to the copyright law, software is a "literary work," but not a "useful object." While this may regrettably be true for some programs, we suggest it is not, in general, a good description of software. The legal notion of "useful object" is important and instructive: copyright is inherently about the expressive (read *artistic*) elements in a work, it does not cover the utilitarian elements, that being the domain of patents. To the copyright scholar, a crucial characteristic of software is its creation in the medium of text; the fact that that text happens to do something useful is irrelevant, because useful behavior is explicitly outside the scope of copyright law. This is so in part because the Constitution set the tone of dividing the world into "Science and the useful Arts" believing there are "authors and inventors" each of whom shall have "the exclusive Right to their *respective* Writings and Discoveries" (article I, Section 8, italics added). Almost exactly 100 years later, a Supreme Court case (*Baker vs. Selden*, 1879) made it explicit that copyright and patents were considered as covering mutually exclusive domains, and indicated that copyright can't cover technology, as that was the intent of patents.

We have attempted to capture some of this conceptual confusion with

[1]For example: *Whelan vs. Jaslow* (1986); *Lotus vs. Paperback* (1990); *Computer Associates vs. Altai* (1992); *Apple vs. Microsoft* (1992); *Lotus vs. Borland* (1990); the 1995 appeals court decision in *Lotus vs. Borland*, and now the Supreme Court's decision on that case.

a pseudo-paradox, pointing out that software is a machine whose medium of construction happens to be text. This captures the inseparably dual nature of software: it is inherently both functional and literary, both utilitarian and creative. This seems a paradox only in the context of a conceptual framework (U.S. law) that cannot conceive of such objects.

What's Important About Software?

To make clear the extent of the mismatch between the legal and technical views, we need to consider both the nature of software and character of the existing intellectual property system. Our observations about software are straightforward; the interesting thing is their consequences. Given their familiarity to this audience, we offer these observations briefly:

- *Programs behave.* They are written to do something.
- *Program behavior and source code text are only loosely connected.* For example, subroutines in the text of a program can be arbitrarily reordered, without having any effect on the behavior of the program. As another example, two programs with different texts can produce identical (external) behavior.
- *Program behavior is valuable.* Behavior must be valuable, otherwise why else would you pay for software in the retail market? In that situation there is, by explicit arrangement, nothing else in the package: you don't get the source code and the standard shrink-wrap license prohibits all sensible ways (disassembly or decompilation) of examining the object code. All you can do is run it; behavior is all you get.
- *Program behavior can be utilitarian or expressive.* Much of the behavior of entertainment (game) software, for example, is expressive. Such expressive behavior is appropriately covered by existing copyright law; we focus here on the problematic part: utilitarian behavior of software.
- *Program behavior can be very complex; programs use conceptual metaphors to organize that behavior.* A spreadsheet program, for instance, contains a great many behaviors; even the earliest versions had more than 100 different things they could do. But when using the program we don't think about a collection of 100 distinct behaviors, the very notion of a spreadsheet organizes and gives coherence to that collection. The situation is similar for word processors (the metaphor of editing text) and the metaphor of the desktop in operating systems.
- *Program metaphors are valuable.* As one obvious example, the spreadsheet metaphor so completely transformed the experience of using a computer that it is widely acknowledged as the first "killer app," one that virtually created the personal computing market.
- *Programs are machines in many senses.* Like machines, they are valued in a large part for their behavior. They could just as well be machines, given the equivalence of hardware and software.
- *Program behavior arises in part from the industrial design of its internals.* For instance, programs work because their inner workings are, like machinery, carefully created, designed, and crafted. The design is possible because of the skilled know-how of the programmer; that skilled know-how is an important part of the value in a program.
- *Progress in software is typically innovative, not inventive.* The character and pattern of technical advance is not uniform across fields [2]. Some fields are characterized by discrete inventions, such as pharmaceuticals, where the discovery and use of one drug generally proceeds independent of others. Other fields, such as aviation and automobiles, involve complex systems of interacting components,

According to the copyright law, software is a "literary work," but not a "useful object."

One essential element of copyright is that the work be expressive, that is, artistic rather than functional.

and are as a result characterized by cumulative advances in which new developments build on features of the existing technology. Software involves complex systems and is fundamentally cumulative. Advances in software are, moreover, more often innovative. Improvements are incremental rather than inventive in the patent sense of nonobvious. Few of the daily, valuable improvements in programs pass the relatively high threshold of inventiveness demanded of patents.

- *Programs often bear much of their know-how on and near their face,* especially as compared to physical products. That is, the good ideas in programs are often evident from simply using them. This is true for many reasons, among them being that software is a tool, and often, to be useful, a tool has to give away its insights. Its utility is often in the way it aids in conceiving of a task (spreadsheets), and to be useful it must convey that conception to the user. Know-how not evident on the face can at times be uncovered through decompilation, hence our use of the phrase "near the face." There are, of course, programs whose know-how is not readily evident, such as complex optimizing compilers, or those with obscure algorithms (Karmarkar's). The point is comparative, not absolute: compared with physical products, software more often bears its know-how on or near the face.
- *Know-how on or near the face is vulnerable to copying.* Perhaps the most famous example is the visit of Steve Jobs to the Xerox Palo Alto Research Laboratory in 1979, for a tour that included a demonstration of Xerox's Alto computer and its graphical user interface (GUI). The visit resulted directly in the use of the GUI in the Lisa and Macintosh. Referring to the GUI, the head of the Xerox Lab subsequently recalled: "To allow Jobs to see the power of the system . . . was a dumb thing to do . . . Once he saw it, the damage was done; he just had to know that it was doable" [4]. The know-how was all there on the face. Know-how on or near the face is immediately accessible to any observer and hence cannot be kept secret.

What's Important About Intellectual Property Law?

The second half of the case for the mismatch between legal and technical views concerns the existing intellectual property system. For our purposes, we can focus on just a few crucial properties of the three standard mechanisms of intellectual property protection.

One essential element of copyright is that the work be expressive, that is, artistic rather than functional. As noted earlier, copyright does not cover useful objects, or even the useful parts of a decorative object.

One essential element of patent protection is that the advance pass the relatively stringent threshold of inventiveness: the advance must be nonobvious to someone familiar with the technology.

One essential element of trade secret protection is that the trade secret remain secret; information emerging into public view via reverse engineering is then available for use by anyone.

The fundamental mismatch is now easily seen:

- If much of the value in software is in its useful behavior, copyright is a fundamentally inappropriate mechanism because it does not protect useful behavior.
- If most software is innovative rather than inventive, most software is not patentable, and therefore most value in software can not be protected using patents. (We believe this to be true despite the abundance of software patents being granted, some significant percentage of which do not appear to pass the test of nonobviousness to an

observer skilled in the art.)
- If software bears much of its know-how on its face, trade secret protection can't work, for we can rarely keep the improvement a secret.

The problem is fundamental: intellectual property law gives us three traditional mechanisms, yet the value in software is not well protected by any of them.

The history of attempts to use copyright to protect software is particularly instructive here. As we have seen, in the copyright world, the fundamental property of software is that it is an original textual work; hence, copyright on software is protection for a program's text. But because text and behavior are relatively independent, protecting text won't protect behavior, no matter how hard we try. And if we do not protect behavior, we fail to protect one of the things at the core of the value in software. Copyright protection is still useful, of course, because there is value in the literal text of a program, but that's only a part of the value.

Should Software be Treated Differently?

If, as we have argued, software falls outside existing intellectual property laws, it is hardly alone. Innovative (rather than inventive) improvements occur in many fields and are at times found "on the face" of the product. In such areas, copying (in the form of reverse engineering and cloning) is a fact of life, yet those areas seem to survive without disastrous effect. Is software different enough to require distinct treatment? We believe software (and its brethren) should be treated differently, and argue that the crucial factor lies in the distinction between *physical goods* and *information goods*, and the evolution of economies from those based primarily on manufactured physical goods to those increasingly based on information products.

In any economy, mass producing something requires a number of processes, including design, manufacturing, large-scale production, distribution, service, and support. Importantly, where physical goods are involved, each of those processes produces some amount of lead time for an innovator. (By lead time we mean the typically brief period during which an innovator has a market niche to itself by virtue of being an innovator.) The creation of a substantially improved automobile engine, for example, requires time to design, time to build the prototype, time to develop large-scale production processes, time to create production facilities, time to create distribution networks, as well as the creation of field service and support providers.

A second-comer wishing to compete in this market will need to go through each of these processes, even if he or she competes purely by copying the innovator's successful (unprotected) product. All of these processes will take time to put in place, producing lead time during which the innovator will have the market niche to itself, enabling it to charge a price that reflects R&D expenses, and thereby fund future innovation.

Now consider the same circumstance when the innovative product is software. Design still takes time, as does manufacturing (coding). Mass production (copying disks) requires considerably less time and capital investment than facilities for physical products. Distribution is similarly faster and less capital intensive, especially with the growing use of soft-

The problem is fundamental: intellectual property law gives us three traditional mechanisms, yet the value in software is not well protected by any of them.

The point is straightforward: software in particular, and information goods generally, have far less inherent lead time because their mass production, distribution, and servicing typically require less time and capital than physical goods.

ware delivery over networks, where the distribution network is already in place (not to mention inexpensive, available 24 hours, and virtually immediate). Service and support require time and capital expense (principally training and maintaining staff), but the time and expense can be substantially lower than with physical products; the support staff can all be in one place, available via phone (hence no branch offices needed), and much of the support can be automated, with the use of fax-back, downloading patches via the Internet, user bulletin boards, and so forth.

The point is straightforward: software in particular, and information goods generally, have far less inherent lead time because their mass production, distribution, and servicing typically require less time and capital than physical goods.

This discrepancy between lead times of information products and physical products is increased further by the tendency of software to bear its know-how on its face. Reverse engineering a physical product (where know-how may take the form of proprietary production processes) can be a time-consuming, and lead-time preserving, activity. But products with their know-how on the face present no such challenge, increasing the disparity between most software and most physical products in inherent lead time.

One additional property of software magnifies the significance of this lack of lead time: software products can often be copied at a much lower cost than their original development expenditure. The most radical example of this is straightforward piracy, where programs that may have taken years and millions of dollars to develop can be copied at essentially zero cost. The phenomenon holds true even for the less radical example of independent cloning—programs whose behavior is intended to mimic a first-comer's, but whose source code is independently created—as in Paperback's independent reimplementation of Lotus's 1–2–3. While there is considerable effort involved here, that effort is still significantly less than that required in the initial research, development, testing, and ongoing evolution of the original product. Paperback, for example, was able to take advantage of the years of development and experience that Lotus had invested, without having to go through the same extensive and costly experience.

The overall issue is that for software (among other products) the competition cost of the product (the second-comer's cost to create a competitive product once the original is available) is often far less than the innovator's research and development cost. Paperback entered the market with a detailed and comprehensive clone of 1–2–3 with a street price of approximately $69, at the same time Lotus was selling 1–2–3 at a street price of roughly $300.

The software marketplace thus has two distinctive factors that produce a significant impact: there is a relative lack of lead time as compared to physical products, and the cost of producing a competing software product is often far less than the cost of innovating that product. Both of these have a significant effect on the innovator's ability to get a return on investment. The inherently shorter lead time reduces the innovator's window during which it can charge a price reflecting the need to recover research and development expense. The wide gap between the competition cost and the R&D cost means that when lead time runs out and the

competitor arrives, the impact on revenue will be quite significant.

The problem is not unique to the software market. It can occur wherever lead time is inherently short and where the ratio of R&D cost to competition cost is large. Note, too, that our claims are fundamentally comparative, not absolute: the issue is the character of software and its market, relative to markets for physical goods.

As a consequence of all this, software is far more vulnerable to a phenomenon we call *comparatively trivial acquisition of behavioral equivalence*, the speedy appearance of cheaper products with equivalent behavior. Behavioral equivalence matters because much of the value of software is in its behavior. Triviality of acquisition matters because trivial acquisition destroys lead time, thereby destroying the innovator's chance to recoup R&D investment. Finally, we say "comparatively trivial" because the second-comer's effort may be far from trivial considered on its own terms (e.g., Paperback's effort in cloning 1–2–3). However, it can be trivial compared to the effort of the innovator.

In the short term this phenomenon is beneficial. The appearance of competitive products is the essence of markets and even the very fast emergence of less expensive rivals is useful to consumers who benefit from lower prices. But in the longer-term innovation can suffer. Who will be willing to invest substantial time, effort, or capital if there is little chance to recover cost, much less profit?

We view this as a form of market failure, that is, a situation in which parties are unable to consummate a mutually beneficial transaction. In this case, the relevant parties are consumers who would be willing to pay for an innovation and software developers who would be willing to create it for a price. The potential of comparatively trivial copying means innovators have no way to ensure that they will be paid. Realizing this, they won't expend effort on innovation. Hence, investment in innovation falls for reasons other than lack of demand for products. We find this concept a central leverage point in confronting the problem of protecting software and suggest that a new framework for protection of software (and other information products) can appropriately be developed around the core concept of avoiding market failure.

Software is far more vulnerable to a phenomenon we call comparatively trivial acquisition of behavioral equivalence, *the speedy appearance of cheaper products with equivalent behavior.*

A New Framework

We begin sketching a new framework by specifying first what we would like the framework to achieve, then suggest what machinery might be needed. This enables discussion to proceed on the question of what we want to accomplish; agreeing on that can clear the way for further work on finding appropriate machinery.

Any framework should be designed to protect the sources of value in software. There is value in the literal code, in the useful behavior produced by the code, in the compilation of applied know-how embodied in the program's internal construction, and in the overall design of the program (its metaphor).

Second, we believe the software marketplace is and will continue to be a breeding ground of innovation. A legal regime for software should seek only to ensure that innovators have the opportunity, in the form of lead time, to test the value of their innovations in the marketplace and reap any consequent reward. The marketplace will offer its unsparing test and issue its final word on worth and value; we seek for innovators only the opportunity for their contributions to be evaluated in the marketplace before they are copied by others.

Third, we suggest that to provide an appropriate scope and duration of protection, the legal regime should be tuned to the "basal metabolic rate" of the market. Markets have a kind of basal metabolic rate that determines the length of product development cycles. For software, it

We believe the legal regime can help avoid market failure by rewarding the innovator, not just the marketer.

may be the one or two years required to create and to test a new product, or the roughly 12- to 18-month interval required to develop and to test a new release of a product. The rate is not arbitrary, but a real empirical phenomenon dependent on a wide range of factors, including the underlying technology (how long it takes to develop and debug a program) and human psychology (how soon consumers will buy an upgrade after purchasing the original program).

Fourth, we believe the legal regime can help avoid market failure by rewarding the innovator, not just the marketer. Xerox PARC may have been the innovator of the desktop metaphor, but Apple was clearly the successful marketer. We suggest everyone benefiting from an innovation should contribute to the costs of R&D.

Machinery for a New Framework

Four mechanisms go a long way toward achieving these goals.

- Traditional copyright protection for literal code;
- Protection against behavior clones for a market preserving period;
- Registration of innovation to promote disclosure and dissemination; and
- A menu of off-the-shelf liability principles and standard licenses.

In protecting the sources of value, the literal code of a program is well taken care of by traditional copyright, hence we begin by building on the existing foundation of copyright, limiting its use to literal program text.

No existing mechanism protects innovative program behavior and design. To remedy this we suggest that cloning of software innovations—copying innovative behavior and design—be restricted for a brief, market-preserving period of time.

One obvious and difficult question is, what's the correct period of time? The authors of this article spent considerable effort attempting to answer this question. After a thorough study of the issue we realized: We don't know. What's equally important here is neither do you.

There is an interesting, important, unsolved question here that is open to empirical investigation: What is (are) the metabolic rate(s) of the software marketplace(s)? How long is long enough for the market to establish the worth of an innovation? The issue need not be the stuff of invective and opinion; we have some real homework to do to come to understand our industry. The authors suspect the right number is far closer to about 2 to 5 years than it is to the 17 years of a patent or the 75 years of a copyright. But suspicions are not the issue, real research is.

If we are to provide lead time by restricting behavior clones, we will inevitably have to make the judgment call of how similar two programs must be in order for one to be a clone of the other. That is, what is similar enough to precipitate market failure? We have developed a metric based on three properties: 1) what was cloned (the extent and significance of the behavior and design overlap); 2) how it was created; and 3) the proximity of the second-comer's product and market.

Market failure is clearly more likely if 1) the amount and significance of the behavior and design taken is substantial (a near complete overlap); 2) the second-comer's development effort is rapid, easy, and highly dependent on the first-comer's product (based on extensive black-box testing of the innovative product); and 3) the products are nearly identical and are to compete in the second market. The converse is also true: The smaller the taking, the less dependent the creation, the less similar the results and less proximate the markets, the less likely it is that a second-comer's borrowing will undermine the ability of an innovator to recoup its R&D expenses and invest in further innovations.

Working out the details of this metric will of course be a substantial effort, but we believe it proceeds from the appropriate foundation.

We also propose a copyright-like registration of software innovations, in order to facilitate automatic licensing. We suggest a mechanism similar to that of the Semiconductor Protection Act (SCPA), which normally provides two years of automatic anti-cloning protection to semiconductor designs; the period can be extended if the design is registered. Under our proposal, a software developer might register a new user interface design or an innovative way of accomplishing some behavior (the innovator would not have to register the product as a whole, as under SCPA). In today's networked environment, an electronic repository of such registered materials would enable licensing on standard terms electronically, thereby reducing transaction costs.

Finally, we are concerned with transaction cost. No approach is workable if it costs too much to use. This is of particular importance in a situation where our explicit goal is to facilitate the workings of the marketplace. Under a traditional intellectual property regime, the negotiations involved in acquiring rights to use someone else's creation can be sufficiently time-consuming as to render the attempt pointless. As one example, a number of planned CD-ROMs were never made simply because of the difficulty and expense of acquiring the necessary rights to each one of the small fragments of works that were to be included [1]. This is a clear example of market failure due to high transaction cost overhead.

We believe transaction costs can be kept modest by basing our system on the notion of liability rules rather than the property rules common to an intellectual property approach. Under a liability rule, the market resembles a cafeteria; all innovations you see are available for sale, all you have to decide is whether they are worth the price indicated. This would entail a form of compulsory license: all innovations would be licensable, for an appropriate royalty. While setting the terms and machinery of such a system would be a substantial undertaking, we argue in detail elsewhere [3] why we believe it can be workable for software.

We also believe that significant benefits accrue from a shift away from a traditional intellectual property view. Perhaps most important for the character of the industry, the use of liability rules fundamentally changes the nature of the decision made by a software developer attempting to compete in the market. In today's property-oriented world, the central question is, *what is legal?*, or, *what can I do and still not get sued?*

In a world of liability rules the question is economic rather than legal. *Is licensing the innovation worth it to me as the second-comer? Will it grow the market for my product sufficiently as to be worth the licensing cost?* If so, then it makes good business sense for me to do so; if not, I should consider the alternatives. As one example, I can wait out the (relatively short) lifetime of the protection for the innovation, then use it for free. Of course I may be losing market share during that time, but that's the point: it's now a market-driven decision, not a question of whether I'm about to step on a legal landmine.

Conclusion

Our approach can be summarized by saying that we propose a law focused on innovative behavior, aimed at avoiding market failure by preserving lead time, and that keeps transaction cost low via a menu of standard liability rules. We focus on behavior and program design because literal text is well taken care of by traditional copyright, and because behavior and the design that produces it are major sources of value in software. We focus on innovation rather than invention because most software is innovative, not reaching the patent standard of invention. We seek to preserve lead time because information products

We propose a law focused on innovative behavior, aimed at avoiding market failure by preserving lead time, and that keeps transaction cost low via a menu of standard liability rules.

There is much to do, but we believe the framework proposed provides a foundation well matched to the realities of the technology, the law, and the marketplace.

(and information-rich products) are particularly vulnerable to market failure from comparatively trivial acquisition of behavioral equivalence. And we work from a menu of liability rules rather than property rules in order to have software companies re-focus their attention on business and economic questions, rather than on legal questions.

There is clearly a considerable amount left to do. What is the metabolic rate of the software market? How many such markets and rates are there? How does the rate change as the markets mature? How should the standard licenses look? How do we ensure that whatever mechanism we propose doesn't calcify?

There is much to do, but we believe the framework proposed provides a foundation well matched to the realities of the technology, the law, and the marketplace.

A Look Further Forward

We end by considering briefly how the software problem illustrates a larger issue. We believe the useful behavior of software is one example of a larger category we term "intangible industrial know-how," the skilled effort produced by experienced engineers and designers of many sorts in many fields. That know-how is not currently well protected under any existing intellectual property regime; it is too utilitarian for copyright and insufficiently inventive for patent. That know-how is often carried on the face of products, rendering trade secret ineffective. As a consequence, the know-how can suffer from the same lead-time erosion and resulting market failure as in software.

In the larger sense, we suggest the appropriate item to be protected in the world of increasingly information-rich artifacts is an industrial compilation of applied know-how, bearing its know-how on its face. A computer program is only one such information-rich artifact; other examples include semiconductor chip designs, databases, and perhaps even gene sequences.

The framework proposed here may work not only for software, but for a wide variety of information products, and information-rich products. As our economies become increasingly information-based, we believe that a new framework is needed for such products.

Our near-term goal is to begin what may prove to be an extended discussion of the kinds of rules and structures best suited to our industry. In the best case, the outcome of such a discussion will be a consensus view of what the community believes to be appropriate. In the longer term the goal is to encourage our industry to take an active role in designing its own economic and legal future. ◻

References

1. Cox, M. Multimedia: In making CD-ROMS, technology proves easy compared with rights negotiations. *Wall Street Journal*, (June 28, 1993), Sec. B, 1.
2. Merges, R. P. Nelson, R. R. On the complex economics of patent scope. *Colum. L. Rev. 839*, 4 (May 1990), 839–916.
3. Samuelson, P., Davis, R. Kapor, M. Reichman, J. A manifesto concerning the legal protection of computer programs. *Colum. L. Rev., 2308, 94* (Dec. 1994), 2308–2431.
4. Smith, D. Alexander, R. *Fumbling the Future.* Morrow and Co., New York, 1988, pp. 241–242.

Abut the Authors:
RANDALL DAVIS (davis@ai.mit.edu) is a professor of Computer Science at MIT and associate director of MIT's Artificial Intelligence Laboratory. **PAMELA SAMUELSON** (samuelson@law.mail.cornell.edu) is a visiting professor of law at Cornell Law School. **MITCHELL KAPOR**(mkapor@kei.com) is Adjunct Professor of Media Arts and Sciences at MIT. **JEROME H. REICHMAN** (jreichma@law.vanderbilt.edu) is a professor of law in the field of contracts and intellectual property, Vanderbilt University.