A MANIFESTO CONCERNING THE LEGAL PROTECTION OF COMPUTER PROGRAMS*

*Pamela Samuelson, Randall Davis, Mitchell D. Kapor, & J.H. Reichman***

* The authors wish to thank the Kapor Family Foundation for its generous support of nearly three years of monthly meetings that enabled the collaboration reflected in this Article. We also thank the Foundation for funding that supported a symposium of intellectual property scholars at Columbia University on April 22–23, 1994, to discuss a preliminary draft. Generous funding for the April Symposium also came from the U.S.-Japan Institute of Vanderbilt University and from the Julius Silver Program in Law, Science, and Technology at Columbia Law School. The authors are grateful to the U.S.-Japan Institute's Director, James Auer, and to Harold Edgar, Director of the Julius Silver Program, for their support, planning efforts, and willingness to cohost and participate in the April symposium. We also wish to thank the participants in the April workshop for their many stimulating comments, especially those who took the time to give us written comments. We give special thanks to Wendy J. Gordon. Her spirited championing led to the organization of the symposium and to the publication of this Article and its companion article. We also specially thank Robert J. Glushko for his substantial support and suggestions during critical stages in the writing of this Article. Lastly, we wish to thank Ethan Stone for his remarkable editing.

** Pamela Samuelson is a Professor of Law at the University of Pittsburgh School of Law. Randall Davis is a Professor of Computer Science and Associate Director of the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Mitchell D. Kapor founded Lotus Development Corp. and designed Lotus 1-2-3. He is currently Chairman of the Board of the Electronic Frontier Foundation. J.H. Reichman is a Professor of Law at Vanderbilt Law School.

Introduction

Virtually all of the voluminous published literature[1] and public debate about the legal protection of computer programs has focused on how existing laws can or should apply to computer programs (e.g., whether copyright protects program user interfaces, whether algorithms are patentable, etc.). As the Office of Technology Assessment recently observed, there has been little normative analysis of the kind of legal protection that would be socially desirable for software and how it might best be accomplished.[2] Such an analysis could articulate what aspects of programs are valuable, to what extent legal protection is needed for different loci of value, and how legal protection of software should be tailored to promote consumer welfare. Four years ago the authors of this Article—two technologists and two lawyers—undertook such a normative analysis. This article reports the results of our collaboration.[3]

In brief, we have concluded that while copyright law can provide appropriate protection for some aspects of computer programs, other valuable aspects of programs, such as the useful behavior generated when programs are in operation and the industrial design responsible for producing this behavior, are vulnerable to rapid imitation that, left unchecked, would undermine incentives to invest in software development. These aspects may need some legal protection against cloning that existing legal regimes cannot provide. Most of the considerable controversy about software protection, within the software industry and the legal com-

1. See generally David Bender, Protection of Computer Programs: The Copyright/Trade Secret Interface, 47 U. Pitt. L. Rev. 907 (1986) (finding little conflict between copyright and trade secret); Donald S. Chisum, The Patentability of Algorithms, 47 U. Pitt. L. Rev. 959 (1986) [hereinafter Chisum, Algorithms] (arguing that algorithms are patentable); Donald S. Chisum et al., LaST Frontier Conference Report on Copyright Protection of Computer Software, 30 Jurimetrics J. 15 (1989) [hereinafter LaST Frontier Report]; Anthony L. Clapes et al., Silicon Epics and Binary Bards: Determining the Proper Scope of Copyright Protection for Computer Programs, 34 UCLA L. Rev. 1493 (1987) (defending application of copyright to software); Dennis S. Karjala, Copyright, Computer Software, and the New Protectionism, 28 Jurimetrics J. 33 (1987) (arguing for software copyright protection linked to copyright's piracy prevention policy); Peter S. Menell, An Analysis of the Scope of Copyright Protection for Application Programs, 41 Stan. L. Rev. 1045, 1050 (1989) (arguing for application of copyright idea/expression merger doctrine to allow standardized interfaces and diffusion of scientific ideas); Arthur R. Miller, Copyright Protection For Computer Programs, Databases, and Computer-Generated Works: Is Anything New Since CONTU?, 106 Harv. L. Rev. 977 (1993) (defending CONTU recommendation to protect software with copyright); Raymond Nimmer & Patricia A. Krauthaus, Copyright and Software Technology Infringement: Defining Third Party Development Rights, 62 Ind. L.J. 13 (1986) (analyzing software copyright case law); A. Samuel Oddi, An Uneasier Case for Copyright than for Patent Protection of Computer Programs, 72 Neb. L. Rev. 351 (1993) (arguing that patents are appropriate for software).

2. See Office of Technology Assessment, United States Congress, Finding a Balance: Computer Software, Intellectual Property and the Challenge of Technological Change 5 (1992) [hereinafter Finding a Balance].

3. "What a long, strange trip it's been . . . ." The Grateful Dead, Robert Hunter, Truckin', *on* American Beauty (Warner Bros. 1970).

munity, has arisen when software developers have tried to use existing legal regimes to protect these kinds of program innovations.[4] The authors have been among those who have opposed efforts to stretch the bounds of existing legal regimes to protect these aspects of programs.[5] In

4. See, e.g., Steering Committee for Intellectual Property Issues in Software, National Research Council, Intellectual Property Issues in Software (1991) [hereinafter NRC Report] (reporting a range of views on software protection issues expressed by members of the software industry at a set of workshops on intellectual property rights); Finding a Balance, supra note 2, at 125–55 (discussing differing views among lawyers on software protection controversies).

5. Randall Davis was the court's technical expert in Computer Assocs. Int'l, Inc. v. Altai, Inc., 775 F. Supp. 544 (E.D.N.Y. 1991), aff'd in part, vacated in part, 982 F.2d 693 (2d Cir. 1992). He has written some articles on legal protection for computer programs that are critical of some legal approaches to understanding programs. See Randall Davis, Intellectual Property and Software: The Assumptions Are Broken, Address Before the Proceedings of the WIPO Worldwide Symposium on the Intellectual Property Aspects of Artificial Intelligence (March 25–27 1991) [hereinafter Davis, Broken Assumptions]; Randall Davis, The Nature of Software and Its Consequences for Establishing and Evaluating Similarity, 5 Software L.J. 299 (1992) [hereinafter Davis, Nature of Software].

Mitchell Kapor has expressed concern about under- and overprotection of software innovations by existing legal regimes. See Computers and Intellectual Property: Hearings Before the Subcomm. on Courts, Intellectual Property, and the Administration of Justice of the House Comm. on the Judiciary, 101st Cong., 1st Sess. 238 (1989) (testimony of Mitchell D. Kapor); see also Simson L. Garfinkel et al., Why Patents Are Bad for Software, Issues in Sci. & Tech., Fall 1991, at 50.

J.H. Reichman has written extensively about existing legal regimes' limited ability to protect what is most innovative and valuable about computer programs. He has suggested a new paradigm for the appropriate protection of program innovations. See, e.g., J.H. Reichman, Computer Programs as Applied Scientific Know-How: Implications of Copyright Protection for Commercialized University Research, 42 Vand. L. Rev. 639 (1989) [hereinafter Reichman, Applied Know-How]; Jerome H. Reichman, Electronic Information Tools—The Outer Edge of World Intellectual Property Law, 24 Int'l Rev. Indus. Prop. & Copyright L. 446 (1993) [hereinafter Reichman, Electronic Information Tools] (conference version published at 17 U. Dayton L. Rev. 797 (1992)); J.H. Reichman, Overlapping Proprietary Rights in University-Generated Research Products: The Case of Computer Programs, 17 Colum.-VLA J.L. & Arts 51 (1992) [hereinafter Reichman, Overlapping Rights].

Pamela Samuelson has argued explicitly for a sui generis form of legal protection of computer programs. See Pamela Samuelson, *Benson* Revisited: The Case Against Patent Protection for Algorithms and Other Computer Program-Related Inventions, 39 Emory L.J. 1025, 1148–54 (1990) [hereinafter Samuelson, *Benson* Revisited]; Pamela Samuelson, Creating a New Kind of Intellectual Property: Applying the Lessons of the Chip Law to Computer Programs, 70 Minn. L. Rev. 471 (1985) [hereinafter Samuelson, Lessons of Chip Law]; Pamela Samuelson, CONTU Revisited: The Case Against Copyright Protection for Computer Programs in Machine-Readable Form, 1984 Duke L.J. 663, 764–69 (1984) [hereinafter Samuelson, CONTU Revisited]. In other articles, she has argued for a narrow scope of copyright protection for computer programs. See, e.g., Pamela Samuelson, Computer Programs, User Interfaces, and Section 102(b) of the Copyright Act of 1976: A Critique of *Lotus v. Paperback*, Law & Contemp. Probs., Spring 1992, at 311 [hereinafter Samuelson, Critique of *Paperback*].

Reichman and Samuelson have participated in a number of briefs amicus curiae of copyright professors in support of defense positions in software copyright cases. See Brief Amicus Curiae of Copyright Law Professors, Lotus Dev. Corp. v. Borland Int'l, Inc., No. 93-2214 (1st Cir. appeal docketed Dec. 14, 1994) reprinted in 16 Hastings Comm. & Ent. L.J.

this Article, we argue that the software controversy has focused on the right problem, but has raised it in the wrong legal milieu. The problem is that behavior and other industrial design elements of programs are often expensive to develop and inexpensive to copy. This makes it possible for an imitator to produce a functionally indistinguishable program—a clone—that the imitator can sell at a lower price. Competition from these clones can destroy incentives to invest in software innovation. None of the existing legal regimes is well-suited to solving this problem.

We are not the first to suggest a sui generis approach to legal protection of computer programs.[6] However, the idea of sui generis protection

---

657 (1994) [hereinafter Borland Amicus Brief]; Brief Amicus Curiae of Eleven Copyright Law Professors, Sega Enters., Ltd. v. Accolade, Inc., 977 F.2d 1510 (9th Cir. 1992) (No. 92-15655), reprinted in 33 Jurimetrics J. 147 (1992) [hereinafter Sega Amicus Brief]. They have also occasionally consulted with defense counsel in software copyright cases.

Because of our opposition to extension of the bounds of existing legal regimes to capture all valuable aspects of software, we have sometimes been characterized as "antiprotectionists" (or worse). See, e.g., Anthony L. Clapes, Software, Copyright & Competition: The "Look and Feel" of the Law 47 (1989) (describing Samuelson's views as technophobic); Anthony L. Clapes & Jennifer M. Daniels, Revenge of the Luddites: A Closer Look At *Computer Assocs. v. Altai*, Computer Law., Nov. 1992, at 11 (characterizing as Luddites those who regard computer programs as utilitarian works); Oddi, supra note 1, at 354 n.3 (characterizing Kapor as an antiprotectionist).

6. At one time, the World Intellectual Property Organization and the government of Japan proposed a sui generis form of legal protection for computer programs. See World Intellectual Property Organization, Model Provisions on the Protection of Computer Software, 14 Copyright 6, 12–13 (1978) [hereinafter WIPO Proposal]; see also Dennis S. Karjala, Lessons From the Computer Software Protection Debate in Japan, 1984 Ariz. St. L.J. 53, 61–70 (describing Japanese proposal). There have also been many articles (apart from those written by this Article's authors, cited supra note 5) proposing a sui generis form of legal protection for computer programs. See, e.g., Rochelle C. Dreyfuss, Information Products: A Challenge To Intellectual Property Theory, 20 N.Y.U. J. Int'l L. & Pol. 897 (1988); Elmer Galbi, Proposal for New Legislation to Protect Computer Programming, 17 Bull. Copyright Soc'y 280, 283–92 (1970); Irwin R. Gross, A New Framework for Software Protection: Distinguishing Between Interactive and Non-Interactive Aspects of Computer Programs, 20 Rutgers Computer & Tech. L.J. 107, 177–86 (1994); Peter S. Menell, Tailoring Legal Protection for Computer Software, 39 Stan. L. Rev. 1329, 1364–67, 1371 (1987); Robert H. Rines et al., Computer Software: A New Proposal for Intellectual Property Protection, 29 J.L. & Tech. 3, 4 (1988); Richard H. Stern, The Bundle of Rights Suited to New Technology, 47 U. Pitt. L. Rev. 1229, 1246–55 (1986); Vance F. Brown, Comment, The Incompatibility of Copyright and Computer Software: An Economic Evaluation and a Proposal for a Marketplace Solution, 66 N.C. L. Rev. 977, 1005–12 (1988); Mary B. Jensen, Comment, Softright: A Legislative Solution To the Problem of Users' and Producers' Rights in Computer Software, 44 La. L. Rev. 1413, 1448–55 (1984); Virginia R. Lyons, Note, Carrying Copyright Too Far: The Inadequacy of the Current System of Protection for Computer Programs, 12 Hastings Comm. & Ent. L.J. 81, 96–98 (1989); John C. Phillips, Note, Sui Generis Intellectual Property Protection for Computer Software, 60 Geo. Wash. L. Rev. 997 (1992); see also Finding a Balance, supra note 2, at 30–31 (discussing three options for protecting computer programs); Office of Technology Assessment, United States Congress, Intellectual Property Rights in an Age of Electronics and Information 81–84 (1986) [hereinafter OTA Report] (suggesting that copyright law cannot be successfully applied to computer programs); John H. Barton, Adapting the Intellectual Property System to New Technologies, *in* Global Dimensions of

for software has generally fallen out of favor since the United States, Japan, and the European Union, among others, decided to use copyright to protect programs.[7] This choice was recently elevated to a norm of international trade when the Trade-Related Intellectual Property Rights (TRIPS) component of the Uruguay Round of revisions to the General Agreement on Tariffs and Trade (GATT) included a provision to provide copyright protection to computer programs.[8]

While the recent Office of Technology Assessment (OTA) report finds that copyright is proving a satisfactory means for protecting program code against exact duplications,[9] it observes that there is a high degree of controversy over scope-of-protection issues[10] and suggests that despite the advantages of incremental accommodation of software by existing law,

> there are questions as to whether this process of accommodation can—or should—continue indefinitely. With respect to software, there may be a point where it becomes preferable to complement or substitute for the existing structures, rather than extend the scope of copyright to fit certain aspects of software—perhaps, cumulatively, at the expense of other types of works.[11]

The Office of Technology Assessment states that it may be easier for Congress to achieve a proper balance in policy objectives through a sui generis approach to software protection than could be achieved through

---

Intellectual Property Rights in Science and Technology 256, 266 (Mitchel B. Wallerstein et al. eds., 1993) [hereinafter Global Dimensions] ("CONTU was almost certainly wrong in its judgment that the copyright system should be used instead of a sui generis approach."); Gerald Dworkin, Copyright, Patents, and/or Sui Generis: What Regime Best Suits Computer Programs?, *in* International Intellectual Law and Policy (Hugh Hanson ed., forthcoming 1995) (recommending keeping open mind about whether sui generis protection might be preferable form of legal protection for software); Allen Newell, *Response*: The Models Are Broken, the Models Are Broken!, 47 U. Pitt. L. Rev. 1023, 1024–34 (1986) (computer scientist offers reasons why a new model of legal protection may be needed for computer software innovations).

7. The United States amended its copyright law in 1980 to extend the protection of this law to computer programs. See Act of Dec. 12, 1980, Pub. L. No. 96-517, 94 Stat. 3015, 3028–29 (codified, as amended, at 17 U.S.C. §§ 101, 117 (1988)). Under pressure from the United States, Japan adopted a copyright approach to the legal protection of software as well, instead of the sui generis approach that it had considered. See Karjala, supra note 6, at 79. In 1991, the Council of the European Communities (as it was then known) adopted a Directive instructing member states to adopt copyright law as a form of copyright protection for software. See Council Directive 91/250, art. 1, 1991 O.J. (L 122) 42 [hereinafter EC Directive].

8. See General Agreement on Tariffs and Trade—Multilateral Negotiations (the Uruguay Round): Agreement on Trade-Related Aspects of Intellectual Property Rights, Including Trade in Counterfeit Goods, 33 I.L.M. 81, 87 (1994) [hereinafter GATT/TRIPs]. For a general discussion of GATT's advantages as a means to deal with worldwide intellectual property piracy, see Marshall A. Leaffer, Protecting United States Intellectual Property Abroad: Toward a New Multilateralism, 76 Iowa L. Rev. 273, 300–03 (1991).

9. See Finding a Balance, supra note 2, at 22.

10. See id. at 150–55.

11. Id. at 8 (emphasis omitted).

use of existing legal regimes wherein changes in scope of protection to accommodate software might distort principles of protection as applied to other categories of works.[12] This Article offers a basis upon which such a complementary or substitute legal regime might be constructed.

We begin in Section 1 by making manifest some characteristics of computer programs that legal discourse about programs has obscured from view.[13] Although programs are texts and their texts can be valuable, the most important property of programs is their behavior (i.e., the set of results brought about when program instructions are executed). Also valuable is the industrial design responsible for producing behavior and the conceptual metaphors that give behavior coherence. Section 2 shows that these aspects of programs are vulnerable to rapid copying because the know-how necessary to construct a functionally equivalent work is on or near the "face" (i.e., the surface) of the product sold in the marketplace. Section 2 will also show that existing legal regimes cannot provide remedies for appropriations of these kinds of program know-how.

Sections 3, 4, and 5 provide foundational concepts upon which to build a new legal regime for the protection of the applied know-how found in the design of program behavior. Section 3 explains why it is desirable to take a market-oriented approach to providing legal protection to these aspects of software. The law should intervene only when it is necessary to avoid the market failure[14] that can arise from rapid copying. Section 4 discusses the evolution of the software market and characteristics of the present industry of which a market-oriented legal regime should be cognizant. Section 5 recommends consideration of three principal factors to judge the likelihood of market failure from rapid copying: (1) the nature of the entity protected, (2) the means by which a second comer acquires access to the innovation and the degree of dependence

12. See id. at 26–27.

13. See, e.g., John A. Kidwell, Software and Semiconductors: Why Are We Confused?, 70 Minn. L. Rev. 533, 535–40 (1985) (suggesting that some of the confusion about applying existing law to software is attributable to confusion about the nature of software).

14. We use the term "market failure" to refer to an other than optimal level of investment in software development that results from the legal underprotection or overprotection of its most valuable innovations. Economists use "market failure" to refer to a variety of situations in which the norms of perfect competition have been violated, including those involving "public goods." See, e.g., Richard G. Lipsey & Peter O. Steiner, Economics 426–27 (6th ed. 1981) (discussing "market failure"). One commonly recognized public-goods problem is the market failure that arises from nonexcludability of information innovations. Although intellectual property rights aim to overcome this public-goods problem, they do so imperfectly. See, e.g., Robert D. Cooter & Thomas S. Ulen, Law and Economics 45–49, 112–16 (1988). Our use of the term "market failure" is consistent with the broader usage. We do not claim that the approach we recommend here would cure all market failures in software or cure them perfectly, but we do think our approach identifies a source of particularly harsh market failures arising from a mismatch of software and existing legal regimes. It provides a basis for avoiding these market failures, with an eye to keeping transaction costs low and to facilitating transfers unlikely to occur under other regimes.

of the second comer's product, and (3) the degree of similarity, both in details and in target market, between the first and second products.

Sections 6 and 7 aim to provide a basis for implementing the market-oriented legal regime whose concepts were discussed in the three previous sections. Section 6 describes a number of design principles and goals that emerged from our discussion about how to construct such a regime. Section 7 describes a number of possible legal mechanisms for implementing a market-oriented approach and uses the design principles suggested in Section 6 as criteria to judge the viability of the various mechanisms considered. The approach that seems to match best with the design principles is one that would provide software developers with a market-preserving period of protection against cloning. We also suggest that developers should have a certain period of time within which to register their program design innovations; registration would provide compensation for the reuse of the innovation by a second comer for a period of time after the expiration of the anti-cloning blocking period. This would mean that second comers would contribute to the research and development costs incurred by innovators.

We do not provide details of a model statute. Rather, we contribute a basic framework for constructing a new form of legal protection for program innovations, hoping to facilitate and direct the political debate which alone could fix the details of implementation. This debate should include those in the software industry, lawyers who serve this industry, economists, and others with a stake in legal protection of software innovations, and should undertake to refine and particularize a legal mechanism.[15] Working from the preferred approach identified in the previous Section, Section 8 discusses three policy options for accepting and implementing this approach.

1 Important Characteristics of Computer Programs

Computer programs have a number of important characteristics that have been difficult for legal commentators and decisionmakers to perceive. First, the primary source of value in a program is its behavior, not its text. Second, program text and behavior are independent in the sense that a functionally indistinguishable imitation can be written by a

15. For example, while we suggest that the term of protection should be short, we have refrained from endorsing any particular term of protection. Our central concern is not with scope of protection as such, but rather with suitability of the protection to the nature of computer programs, the innovations they embody, and the software marketplace. Choosing a particular period of protection should be the end result of the public debate that we are merely trying to frame in meaningful terms. Suggesting a particular period of anti-cloning protection is irrelevant to, and would distract readers from, the analysis which leads us to suggest that there should be some anti-cloning protection in the first place. The difficulty of tailoring intellectual property rights to achieve the proper balance of incentives to avoid both under- and overprotection of innovation is well known. See, e.g., Cooter & Ulen, supra note 14, at 135–49.

programmer who has never seen the text of the original program. Third, programs are, in fact, machines (entities that bring about useful results, i.e., behavior) that have been constructed in the medium of text (source and object code). The engineering designs embodied in programs could as easily be implemented in hardware as in software, and the user would be unable to distinguish between the two. Fourth, the industrial designs embodied in programs are typically incremental in character, the result of software engineering techniques and a large body of practical know-how.

1.1 Programs Behave

1.1.1 Computer Programs Are Not Only Texts; They Also Behave

At first glance, a program appears to be a textual work. Source code[16] is clearly some form of text, even if in a strange language not easily read by the casual observer. The view of programs as texts has been widely adopted in the legal community.[17]

While conceiving of programs as texts is not incorrect, it is seriously incomplete. A crucially important characteristic of programs is that they behave; programs exist to make computers perform tasks.[18] Program behavior consists of all the actions that a computer can perform by executing program instructions. Among the behaviors commonly found in word processing programs, for example, are copying text, deleting text, moving text from one place to another, and aligning margins. Program manuals typically contain a good description of much of a program's behavior.[19] Programs often compete on the basis of behavior; advertisements routinely list the capabilities (i.e., "features" which are discrete units of behavior) a program has that its competitors do not.[20] Advertisements for tax preparation programs, for example, may emphasize the va-

16. After the design for a program has been completed, programmers typically write out (in a computer language such as FORTRAN, BASIC, C, or Pascal) the set of statements or instructions that will constitute the program. This is known as "source code." Computers, however, cannot run source code; the code must first be converted to a machine-executable form called "object code." This is done with the use of programs, known as compilers or assemblers. For a technically accurate discussion of source and object code in the legal literature, see, e.g., Samuelson, CONTU Revisited, supra note 5, at 672-89.

17. See, e.g., Nat'l Comm'n on New Technological Uses of Copyrighted Works, Final Report 9-10 (1978) [hereinafter CONTU Report]; Clapes et al., supra note 1, at 1511.

18. The copyright law implicitly recognizes this in its definition of computer program: "A 'computer program' is a set of statements or instructions to be used directly or indirectly in a computer *in order to bring about a certain result.*" 17 U.S.C. § 101 (1988) (emphasis added); see Newell, supra note 6, at 1029, 1032 (pointing out importance of program behavior).

19. See, e.g., User's Guide, TouchBase Pro 3-6 (1993).

20. See, e.g., Advertisement for IBM OS/2, PC Week, June 20, 1994, at 62 (comparing features of OS/2 and Windows 3.1). Features of computer programs are discussed infra notes 304-312 and accompanying text.

riety of tax forms they can handle (that is, the variety of tax-preparation behaviors the program is capable of producing).

Behavior is not a secondary by-product of a program, but rather an essential part of what programs are. To put the point starkly: No one would want to buy a program that did not behave, i.e., that did nothing, no matter how elegant the source code "prose" expressing that nothing.

Hence, every sensible program behaves. This is true even though the program has neither a user interface,[21] nor any other behavior evident to the user. When someone sends electronic mail, for example, she will interact with a program that initiates transmission of the mail. This program hands the message off to a sequence of other programs that see to its delivery in a manner that is invisible to the user. The transmission programs have neither user interfaces nor visible behavior. Nonetheless, each behaves in ways important to the user.

1.1.2 Text and Behavior Are Largely Independent

Although the text of a program is designed to produce certain behaviors, program text and behavior are more independent than might be readily apparent. That is, two programs with different texts—e.g., VP-Planner and Lotus 1-2-3—can have completely equivalent behavior.[22]

A second comer can develop a program having identical behavior, but completely different text through a process sometimes referred to as "black box" testing.[23] This involves having a programmer run the program through a variety of situations, with different inputs, making careful notes about its behavior. A second programmer[24] can use this descrip-

21. A user interface is that part of a program's behavior that exists at the boundary between the program and the outside world. It is akin to the skin or a membrane of an organism. The user interface is the means by which the user and the program interact to produce the behavior that will accomplish the user's tasks, much as organs of touch, sight, and hearing are the means by which human organisms interact with their environment. Just as an organism's behavior is conditioned upon, but is not equivalent to, the functioning of its senses, the user interface of a program is not the same thing as the program's overall behavior. Some legal commentators have been confused about the distinction between a program's behavior and its user interface. See, e.g., Miller, supra note 1, at 1007–08.

22. Two widely known examples of this phenomenon were the VP-Planner program developed by Paperback Software and The Twin program developed by Mosaic Software. The behavior of each was virtually identical to that of Lotus 1-2-3, a feature VP-Planner boasted of in its manual. See Lotus Dev. Corp. v. Paperback Software Int'l, Inc., 740 F. Supp. 37, 69–70 (D. Mass. 1990). Neither imitator had any access to the Lotus source code.

23. See, e.g., Duncan M. Davidson, Common Law, Uncommon Software, 47 U. Pitt. L. Rev. 1037, 1080–84 (1986) (discussing the black box metaphor and a model for how it might be used in resolving software copyright disputes; under Davidson's model, "black box testing," i.e., examining a program in operation, would be lawful).

24. "Clean-room" development involves having two teams of programmers, one of which operates in a "clean room" (without access to or knowledge of the text of the program being analyzed). The "dirty room" team of programmers analyzes another firm's program in order to extract information about it, such as information about its interface.

tion to develop a new program that produces the specified behavior (i.e., functionally identical to the first program) without having access to the text of the first program, let alone copying anything from it. A skilled programmer can, in other words, copy the behavior of a program exactly, without appropriating *any* of its text.

The independence of text and behavior is one important respect in which programs differ from other copyrighted works. Imagine trying to create two pieces of music that have different notes, but that sound indistinguishable. Imagine trying to create two plays with different dialogue and characters, but that appear indistinguishable to the audience. Yet, two programs with different texts can be indistinguisable to users.

1.1.3 Behavior is Valuable

Behavior is not the only source of value in a program, but it is the most important. People pay substantial sums of money for a program not because they have any intrinsic interest in what its text says, but because they value what it does and how well it does it (its behavior). The primary proof that consumers buy behavior, rather than text is that in acquiring a program, they almost never get a readable instance of the program text. They acquire object code[25] not source code, a text that is, at best, quite obscure. Even so, shrink-wrap licenses purport to prohibit all sensible ways of examining that text (i.e., disassembling or decompiling the object code).[26] Hence, when buying software in the retail market, consumers buy behavior and nothing more. There is, by explicit arrangement, nothing else in the package.

This team passes the information on to the "clean room" group of programmers who use that information to reimplement the functionality. See NRC Report, supra note 4, at 77-78. Because the "clean room" team has not had access to the text of the other program, similarities in details of the two programs will generally be attributed to implementation of the same functionality. See, e.g., NEC Corp. v. Intel Corp., 10 U.S.P.Q.2d (BNA) 1177 (N.D. Cal. 1989); see also Gary R. Ignatin, Comment, Let the Hackers Hack: Allowing the Reverse Engineering of Copyrighted Computer Programs to Achieve Compatibility, 140 U. Penn. L. Rev. 1999 (1992).

25. See supra note 16 for a discussion of object code.

26. Disks containing computer programs are often sold in a box covered with a shrink-wrap. Many software developers put a printed form underneath the plastic sheeting which advises prospective purchasers that they are only licensing the software and that this license is subject to many restrictions. Such forms also tend to state that by opening the package, the person will have agreed to be subject to all of the stated restrictions. A number of commentators have questioned the validity of these shrink-wrap licenses, both as a matter of contract law and as a matter of intellectual property policy. See, e.g., Thomas L. Hazen, Contract Principles as a Guide for Protecting Intellectual Property Rights in Computer Software, 20 U.C. Davis L. Rev. 105, 112 (1986); David A. Rice, Licensing the Use of Computer Program Copies and the Copyright Act First Sale Doctrine, 30 Jurimetrics J. 157 (1990). Decompilation and disassembly of program code is discussed infra notes 89-91, 326-337 and accompanying text. A draft proposal to validate many provisions of software shrink-wrap licenses is currently under consideration. See U.C.C. § 2-2203 (Draft Sept. 10, 1994).

Not only do consumers not value the text that programmers write (source code), they do not value any form of program text, not even object code. Computer science has long observed that software and hardware are interchangeable: Any behavior that can be accomplished with one can also be accomplished with the other.[27] In principle, then, it would be possible for a purchaser of, say, a word processor to get a piece of electronic hardware that plugged into her computer, instead of a disk containing object code. When using that word processor, the purchaser would be unable to determine whether she was using the software or the hardware implementation.

Thus, to the user of a word processor, it is immaterial whether the program is implemented in hardware or software. The user would just as willingly buy the hardware as the software version, as long as it exhibited the desired behavior. When buying the hardware version, there would be no text in the package at all (not even the obscure text of object code). Yet the value to the user would be unchanged.

All this stands in sharp contrast to traditional literary works which are valued because of their expression (i.e., what they say and how well they say it). Programs have almost no value to users as texts. Rather, their value lies in behavior.

1.1.4 Programs with Identical Behavior Can Be Market Substitutes

If behavior is the primary source of value in a program, then the potential for two programs to be behavioral equivalents is of considerable competitive significance. Two programs that produce identical behavior may, from the standpoint of consumers, be perfect market substitutes.[28]

The lawsuits concerning VP-Planner and The Twin illustrate this point. There, the ability to create a program with identical behavior was far more than a laboratory curiosity. The developers of these programs hoped that their "work-alikes" would take some of Lotus's share of the market for electronic spreadsheet programs. Lotus sued Paperback and Mosaic for infringement, hoping to forestall this erosion.[29]

To a consumer, textual differences among programs are typically neither discernible nor material to a purchasing decision. Hence, programs with identical behavior are market substitutes even if they have different text. This point will be of particular significance when we discuss

27. See, e.g., Terrence W. Pratt, Programming Languages: Design and Implementation 19 (2d ed. 1984).

28. Factors that may cause consumers not to regard clone software as perfect substitutes include the reputation of the innovator for reliability, the availability of service, and access to updates, among other things.

29. Lotus won its lawsuit against Paperback, see Lotus Dev. Corp. v. Paperback, 740 F. Supp. 37 (D. Mass. 1990), as well as against Mosaic. Paperback settled before appeal. See Lotus Settles Copyright Case, N.Y. Times, Oct. 18, 1990, at C4. Mosaic went into bankruptcy after Lotus won at the District Court level. See Lawrence Edelman, Lotus Halts Action in 'Twin' Sales, Boston Globe, Apr. 3, 1991, at 30.

whether copyright, which focuses on the text, is an optimal form of legal protection for software.

1.2 Programs Are Machines Whose Medium of Construction Is Text

Traditional literary works, such as books, do not behave. Programs, like other machines, do. Programs have a dual character because they are textual works created specifically to bring about some set of behaviors. We attempt to capture this intriguing dual nature of software by describing software as a machine whose medium of construction happens to be text.[30]

To say that software is a machine is not to make an abstract metaphorical statement. Computer programs and physical machines have more in common than might be suggested by the legal description of programs as texts. First, behavior is common to both of them. Physical machines (e.g., automobiles, televisions) produce a variety of useful behaviors, and so it is with programs. They exist in order to "bring about a certain result,"[31] i.e., some behavior.[32]

Second, as noted above, programs can just as well *be* physical machines; an electronic device that plugged into the computer could deliver identical behavior.[33] In any given situation there may be reasons, such as cost, that motivate developers to embody behavior in one form or the other, but it is important to note that programs and physical machines are completely interchangeable in the sense described.[34]

Third, programs, like other machines, often work together with other programs (and with other machines) to bring about their results.

30. See Davis, Broken Assumptions, supra note 5, at 111.

31. This phrase is taken from the copyright statute's definition of computer program, see 17 U.S.C. § 101, but it applies equally well to other kinds of machines.

32. Traditional texts may tell humans how to do a task; they do not, however, perform the task.

33. See supra note 27 and accompanying text.

34. We do not mean simply providing the software in a hard-wired form like read-only memory (ROM). In principle, the behavior of any program can be duplicated exactly by a machine built from basic electronic components like transistors, AND gates, OR gates, and so forth. So far, this equivalence has had modest commercial impact because mimicking even a medium-size software program would require a very large collection of hardware. Hence, the conversion rarely makes economic or engineering sense. Even so, some patents have exploited it. For example, one patent concerns a means of controlling access to data files by computer users. The patent describes a device constructed from digital hardware, but notes revealingly: "To those skilled in the computer art it is obvious that such an implementation can be expressed either in terms of a computer program (software) implementation or a computer circuitry (hardware) implementation, the two being functional equivalents of one another . . . . For some purposes a software embodiment may likely be preferable in practice." U.S. Patent No. 4, 135, 240 to Ritchie, issued Jan. 16, 1979, at col. 5, line 48 (Protection of Data File Contents). In fact, this particular "device" is (and has been) a part of every computer's operating system, and as such, as the patent author surely knew, would never be found in a hardware embodiment. The author simply used the equivalence of the two to gloss over what was apparent to any technical reader, and thus patented a piece of software as if it were a hardware device.

Application programs, for example, call upon operating system programs, which in turn call upon microcode programs for the execution of program functions.

To enable a program to interact with other programs, a programmer must make sure that it sends and receives signals in the manner required by those programs.[35] One of the critically important tasks of software development is designing this information flow to allow interoperation.[36] Software developers refer to this as designing the program's "interface." Software interfaces are the information equivalents of the gear teeth, levers, pulleys, and belts that physical machines use to interoperate.

Fourth, creating programs is a process of building and assembling functional elements. Where physical machines are built from physical structures like gears, wires, and screws, programs are built from information structures, such as algorithms and data structures.[37] In software, these components must work together in a very carefully orchestrated way that resembles nothing so much as an intricate mechanical device consisting of thousands of delicate gears and levers. In a mechanical device, the gear teeth must mesh exactly, and levers must move at just the right moments. If a single component of either a hardware or software machine fails to work as intended, disastrous effects can propagate through the rest of the machine in the blink of an eye. It is not an accident that a program that fails is said to have "crashed."

Fifth, like physical machines, programs are often large and complex.[38] Programs with hundreds of thousands of lines are common in industrial applications. The largest programs consist of several million lines of code.[39] By comparison, a very complex mechanical device—such as Boeing's new 777 airliner—is composed of a few hundred thousand distinct parts, with perhaps three million individual parts in total (count-

35. See, e.g., NRC Report, supra note 4, at 51–54.

36. See id. at 52 ("Software interfaces are often the product of significant investment, creative activity, and engineering effort."); see also Frederick P. Brooks, Jr., No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, Apr. 1987, at 10, 12 (describing the complexity of conforming a program to other systems' interfaces).

37. An algorithm is "a prescribed set of well-defined, unambiguous rules or processes for the solution of a problem in a finite number of steps"; data is "a formalized representation of facts or concepts suitable for communication, interpretation, or processing by people or by automatic means"; a data structure is the structure of relationships among data items. See Websters' New World Dictionary of Computer Terms (3d ed. 1988).

38. See, e.g, Brooks, supra note 36, at 11.

39. The operating systems for the IBM AS/400, which appeared in 1988, had 6.9 million lines of source code. See Edward Bride, Software Magazine: A Look Back, 11 Software Mag. 89 (1991). Windows NT, a follow-on operating system to Windows 3.1, contained 4.3 million lines of code in May 1993 while still under development. See G. Pascal Zachary, Climbing The Peak: Agony And Ecstasy Of 200 Code Writers Beget Windows NT, Wall St. J., May 26, 1993, at A1.

ing each nut and bolt).[40] Hence, the largest programs are roughly comparable in component count to some of the most complex mechanical devices.

Although the number of components may be comparable, however, building a program is generally a much more complex task than building a physical machine. Unlike the designer of a physical machine, the software engineer has no standard components. She cannot buy off-the-shelf parts or sub-assemblies from suppliers. Typically, a programmer writes every line of code afresh, no matter how large the program is, or how common its tasks.[41] To perceive the impact of this lack of standard building blocks, imagine trying to design an entire car in complete detail, down to the last fastener, without being able to assume the existence of any standard parts at all (not even nuts, bolts, or screws). With programs, every individual part has to be designed from scratch, and all of their design elements must be tested under a variety of conditions to be sure

40. See Boeing Commercial Airplanes Group, Boeing 777 Fact Sheet (132,500 unique parts, over three million total parts) (on file with the Columbia Law Review).

41. This is not a desirable state of affairs. It is one of the things that makes building large programs difficult. Still, despite progress over the years, the standardized parts available to software engineers are still relatively few and primitive.

Some progress has been made over the years in programming languages, as developers have created "higher-level" languages. Programming languages are characterized in terms of their "level," i.e., the extent to which the terms of the language require the programmer to consider properties of the physical machine. The lowest level language used to create applications is "assembly language," which requires the programmer to know and keep track of such details of the hardware as how many different registers (places to store intermediate computations) there are in the machine. In the earliest years of the industry many programmers wrote their applications in assembly language, creating an individual instruction for each of the program's operations.

"Higher-level" programming languages like FORTRAN (which dates back to the late 1950s), BASIC, and C contain a few basic tools. The higher-level programmer must describe each computation, but need not consider how a specific computer will implement them. Higher-level languages sharply reduce the level of detail in programming, and offer advanced data and control structures. This speeds up coding considerably. However, even in higher-level languages, there is rarely, if ever, any standardized source-code available, and creating an application still involves designing and building all the needed functionality from scratch.

There has been some progress beyond "higher-level" languages. PC operating systems offer more functionality to the application programmer. Packages like X-Windows and Visual Basic provide convenient platforms for building graphical interfaces. The trend toward "object-oriented" programming may eventually produce standard program components. For the moment, however, there are virtually none. See, e.g., Brooks, supra note 36, at 14 (expressing doubts that object-oriented programming will bring about substantial increases in software productivity).

Brooks argues that complexity is "an essential property" of software, and that there is "no single development, in either technology or management technique, that by itself promises even one order of magnitude in productivity, in reliability, in simplicity." Id. at 10–11. Whether this is true or not, progress to date has been slow and painful, and the applications programmer still designs almost every required nut and bolt.

they work.[42] This is one reason why programs are often so complex and their design so difficult.

Thinking of software as a machine makes it clear that source code is the medium in which a program is created, even though the value in a program, as with other machines, lies in its behavior. The behavior of software, like the behavior of other machines, can be either utilitarian or fanciful. The behavior of most machines is utilitarian. Lawn mowers, water pumps, and televisions, for example, all perform useful tasks, as do word processors and spreadsheets. Some machines, however, have purely fanciful behavior. Kinetic sculptures, for instance, are assemblages of mechanical components carefully interconnected to produce certain behaviors that have no utilitarian purpose. Some physical machines produce both utilitarian and fanciful behavior. Elaborate clocks of the seventeenth century not only kept time (a utilitarian task), but announced its passage with an elaborate and artistic dance of human figures. Similarly, some game programs create unusual worlds inhabited by artfully-created characters. Software, like other media, can be used to construct machines with utilitarian behavior as well as machines with fanciful behavior.

Program text is, thus, like steel and plastic, a medium in which other works can be created. A device built in the medium of steel or plastic, if sufficiently novel, is patentable; an original sculpture built of steel or plastic is copyrightable. In these cases, we understand quite well that the medium in which the work is made does not determine the character of the creation. The same principle applies to software. The legal character of a work created in the medium of software should no more be determined by the medium in which it was created than would be a work made of steel or plastic. In this respect, it makes no more sense to talk about copyrighting programs than to talk about copyrighting plastic or steel; it confuses the *medium of creation* and the *artifact* created. In the case of software, the artifact created is some form of innovative behavior, whether utilitarian or fanciful.

All of this has implications for selecting a protection mechanism appropriate to software.[43] Much of the legal commentary concerning

42. The lack of available component parts for software means complexity of another kind. A computer manufacturer, for example, that buys a power supply in the market receives not only the physical part (which it therefore does not have to design and build independently), but also clears itself of obligations to any patent owner for royalties, because a license to use any patents will be built into the purchase price. The situation is quite different with software, where virtually all components must be built from scratch. A software designer takes the risk that one of the thousands of small components he or she has designed and implemented in building a large program will be covered by a patent. All industrial designers face this problem, but with software, the task of checking for patent infringement is quantitatively different: Consider the vastly increased difficulty of checking automobiles for patent infringement if all automobile components were independently designed and constructed, rather than assembled from available parts.

43. See infra notes 156–171 and accompanying text.

software has conceived of it as text and hence taken literary works as the predominant metaphor. We suggest that programs should be viewed as virtual machines and that this has interesting consequences for the proper form of protection.

1.3 Many Programs Rely on Conceptual Metaphors to Organize Behavior

Programs often create a new conception of the tasks they accomplish by providing a metaphor for engaging in the task. The metaphor often enables the creation of new kinds of objects that behave in interesting ways, thereby bringing about a new, synthetic reality, which we term a "virtuality."[44] One of the best known of these metaphors is the word processor.[45] Word processing programs use the conceptual metaphor of paper to provide users with the illusion that they are working with paper (what we might call "virtual paper"). Yet they also extend the concept of paper because word processing paper can do some things that ordinary paper cannot. On ordinary paper, insertions or deletions of text can be difficult and messy. On word processing paper, the old text obligingly moves over to make room for the new words or closes ranks to fill in a gap left by a deletion. Cutting and pasting a chunk of text on ordinary paper produces a fragmented document; word processing paper instantaneously heals from the surgery. Making a minor change to a paper document (e.g., changing pagination) may mean retyping the entire work. On word processing paper, one can easily make incremental changes. The user simply makes the change, and the "paper" takes care of the rest.

The behavior of a well-designed program has an overall coherence that often derives from its conceptual metaphor.[46] By coherence, we mean that the whole is more than a simple juxtaposition of the constituent behaving parts. It is instead a well-orchestrated collection that produces a comprehensible overall behavior. Word processing software, for example, is far from an arbitrarily chosen compilation of behaviors. The notion of a special kind of paper is the basic metaphor that drives and organizes the program's behavior so that it produces a coherent image to the user.

44. See, e.g., NRC Report, supra note 4, at 55–56. User interface designer Bruce Tognazzini speaks of the goal of user interface design as " 'creating an illusion that does not break down.' " Id. at 55.

45. Another well-known example is the desktop metaphor for defining the work space upon which users work with computers. See Bill Curtis, Engineering Computer "Look and Feel": User Interface Technology and Human Factors Engineering, 30 Jurimetrics J. 51, 60–61 (1989).

46. As the NRC Report notes, "the behavior and the visual appearance of the objects in the illusion created on a computer screen, be it a spreadsheet display or a flight simulation, must mesh perceptually with all applications that use the interface." NRC Report, supra note 4, at 55 (paraphrasing Bruce Tognazzini).

Coherence is a valuable element of behavior, particularly for users. Coherent behavior can help the user anticipate what to expect and understand what the program has done or is doing. Coherence in program behavior often results from the conceptual metaphors the program uses. Thus, conceptual metaphors are valuable as organizing principles for program behavior, as well as for the virtual worlds and objects they create. Much of the value in a spreadsheet program, for instance, arises from the rich metaphor it provides for organizing or thinking about the task. Even referring to it as a spreadsheet is revealing: Using it does not make us think we are using software, but rather that we are using a particular tool that enables us to have a particular kind of experience.

Conceptual metaphors are also significant because they can change the nature of and the user's experience of the task.[47] Cutting and pasting (in a virtual sense) is so easily accomplished in a word processor that one soon begins to think of the text as a physical entity that can be picked up and moved around. A sentence, a paragraph, or an entire chapter now becomes a portable object, rather than letters wedded to a single location on the page. Word processing paper thus offers a virtuality in which text and ideas can be rapidly and repeatedly organized and reorganized. The experience of incremental editing is similar. Changing a word and redoing the document is no longer a daunting undertaking. Instead, we routinely go through dozens of drafts.

Spreadsheet programs illustrate even more starkly the power that conceptual metaphors have to change the user's experience of the task. Foremost among the conceptual metaphors of spreadsheet programs is the notion that tables of numbers are "alive." The software recalculates sums in response to different data entries, where a paper spreadsheet merely provides a passive framework for recalculation. Use of a spreadsheet now means only having to describe the relationship between entries in the table (e.g., that one entry should be the sum of the numbers above it) and entering the data.[48] The automated spreadsheet metaphor has so fundamentally changed the experience of using a computer that users feel they are using a spreadsheet, not just a computer.

The value attributable to powerful conceptual metaphors in programs can be made in more objective terms as well. The first automated spreadsheet program was VisiCalc which appeared in 1979. In the fifteen years since, numerous firms have gone into business making spreadsheet programs, and millions of copies of spreadsheet programs have been

47. Indeed, the conceptual metaphors can be powerful even if they do not rely on pre-existing physical entities, such as paper. One user interface designer speaks of user interface designers as "illusionists who, unconstrained by their medium, create their own natural laws." NRC Report, supra note 4, at 55.

48. Before the advent of spreadsheet programs, someone who wanted to use a computer to do calculations had to write a program for each calculation. Spreadsheet programs allow users to accomplish complex calculations without knowing how to program.

sold.[49] Thus, a new metaphor is not merely a way to organize old tasks, it can be a powerful tool that qualitatively changes the task. An innovative conceptual metaphor is one of the most valuable types of software innovation. Spreadsheet programs, for instance, were the class of application program that spurred the first powerful surge in acquisitions of personal computers.[50] The power of this conceptual metaphor created a revolution in the use of computers.

The legal regime that protects software should find some way to protect the effort that produces such valuable new tools. In the current regime this does not appear likely. Although conceptual metaphors are important sources of innovation in software products, they are remote from the program text. Partly because of this and partly because of their abstract character, they would likely be regarded as unprotectable by copyright law.[51]

1.4 Programs Are Industrial Compilations of Applied Know-How

1.4.1 Program Construction Requires Selection and Arrangement of Useful Components

Even programs that employ imaginative conceptual metaphors must be carefully constructed by the programmer. This construction process involves selecting and arranging components so that the software machine produces the desired behavior. Computer programs are inherently compilations of sub-components. The most obvious evidence of this is that programs are built from programs. Programmers routinely work by decomposing large tasks into smaller sub-tasks and sub-sub-tasks. They write a sub-program to accomplish each of the smaller tasks, then orchestrate the sub-programs' interaction so that the combination works together to accomplish an overall task. This innocuous-sounding idea is an important technical notion, key to understanding software development. It is used widely and is essential in building any nontrivial program.

Crucially, there is no material difference between a "task" and a "subtask." The prefix indicates only the relation between them. Thus, pro-

49. See International Data Corp. Report IDC #7862, Professional Support Tools: 1993, Worldwide Markets and Trends 20 (1993) (reporting $1.226 billion in spreadsheet and other business tool software produced by Microsoft, Lotus, and Borland).

50. VisiCalc was a compelling application—an application so important that it, alone justified the computer purchase. Such an application was the last element required to turn the microcomputer from a hobbyist's toy into a business machine. No matter how powerful and brilliantly designed, no computer can be successful without a compelling application. To the people who bought them, mainframes were really inventory machines or accounting machines, and minicomputers were office automation machines. The Apple II was a VisiCalc machine.

Robert X. Cringely, Accidental Empires 64 (1992).

51. See LaST Frontier Report, supra note 1, at 29 (describing conceptual metaphors as "ideas").

grams are built from programs. Many programs on the market today illustrate this clearly: they are composites of programs, each of which could exist separately. Spreadsheets, for example, typically combine calculation, graphing, and database components (among others), each of which is itself a useful program.

Programs are compilations in another sense as well. Even the smallest sub-program is also a compilation of sub-components. Programmers construct sub-programs by assembling into a coherent whole such discrete program elements as data, data structures, and algorithms.[52] The "engineering" in software engineering involves knowing how to assemble these components to produce the desired behavior.

1.4.2 Programs Are Compilations of Useful Behavior

Because programs are inherently composite in character (larger programs are built from smaller programs) and because programs behave, we say that programs are compilations of behavioral components. That is, a program is fundamentally a compilation whose constituent elements are behaviors.

Each of the behaviors is, in turn, carefully designed so that, collected, they work together to produce the overall desired behavior. Every sub-program captures a small insight about what behavior is needed and how to accomplish it. Every sub-assembly of components captures insights about how smaller behaviors can combine to produce the desired overall behavior. Hence programs are not just compilations of behavior, they are compilations of useful behavior, a carefully selected collection of utilitarian components working together.[53]

1.4.3 Constructing Programs Is an Industrial Design Process

Once one understands that programs are machines that happen to have been constructed in the medium of text, it becomes easier to understand that writing programs is an industrial design process akin to the

52. See supra note 37.

53. This is true even when the program produces behavior that is fanciful in character. A program that creates an image on a screen, for instance, must be carefully designed to do this task, no matter how fanciful the resulting image, just as would a mechanical device designed to create the same image on paper. The mechanical device would have to be carefully designed and built to ensure, among other things, that the ink ends up on the paper (rather than on the floor), that it is dispensed in appropriate amounts in the appropriate places, etc. So it is for a program to do the same task: like any machine, it behaves, and the behavior must be carefully designed and constructed in order to put the appropriate bits of color on the screen at appropriate places. Hence the program that creates a fanciful work is still a machine, and that machine is quite distinct from the fanciful work created; a mechanical device hardly becomes a work of art simply because it is capable of producing a work of art, any more than a human author becomes a literary work as a result of having written a novel.

design of physical machines.[54] Each stage of the development process requires industrial design work: from identifying the constraints under which the program will operate, to listing the tasks to be performed (i.e., determining what behavior it should have), to deciding what component parts to utilize in bringing about this behavior (which in the case of software, includes algorithms and data structures), to integrating the component utilitarian elements in an efficient way.[55]

A substantial amount of the skilled effort of program development goes into the design and implementation of behavior.[56] Designing behavior involves a skilled effort to decompose the overall, complex task (e.g., word processing) into a set of simpler sub-tasks requiring simpler behaviors (e.g., deleting a character). Constructing the interaction of these sub-tasks to produce useful behaviors (e.g., deciding whether the "delete word" command should be implemented as a sequence of delete character commands) also requires design skill. Knowing how and where to break up a complicated task and how to get the simpler components to work together is, in itself, an important form of the engineering design skill of programmers.[57]

The goal of a programmer designing software is to achieve functional results in an efficient way.[58] While there may be elements of individual style present in program design, even those style elements concern

54. For in-depth discussions for legal protection for industrial design, see generally, J.H. Reichman, Design Protection after the Copyright Act of 1976: A Comparative View of the Emerging Interim Models, 31 J. Copyright Soc'y U.S.A. 267 (1984) [hereinafter Reichman, Design Protection after 1976] (arguing for more comprehensive design protection); J.H. Reichman, Design Protection in Domestic and Foreign Copyright Law: From the Berne Revision of 1948 to the Copyright Act of 1976, 1983 Duke L.J. 1143 [hereinafter Reichman, Design Protection 1948–1976] (discussing different approaches to protecting designs).

55. See, e.g., NRC Report, supra note 4, at 45–46 (describing Dan Bricklin's model of the software development process); Brooks, supra note 36, at 11 ("The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. . . . I believe the hard part of building software to be the specification, design, and testing of this conceptual construct . . . .") (emphasis omitted).

56. For articles emphasizing the importance of design in the development of software, see Brooks, supra note 36, at 18; Mitchell Kapor, A Software Design Manifesto, 16 Dr. Dobbs J. 62 (1991).

57. See, e.g., NRC Report, supra note 4, at 45 (" '[It] isn't the programming that is hard; it is figuring out what we're trying to do that is hard.' ") (quoting Randall Davis).

58. The software copyright case law has recognized this point. See, e.g., Computer Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 708 (2d Cir. 1992). It is worth noting that efficiency must always be measured in terms of the programmer's goals. A design that is highly efficient in conserving memory may not be highly efficient in speed of execution. There is, as a consequence, no one "most efficient" design for a software program. Any well-designed program is as efficient in relation to its design goals as its developers have been able to make it, given their skill and the constraints under which they were operating. Furthermore, the case law has recognized that two programmers working separately on the same industrial design problem in software may happen upon the same efficient solution. For this reason, the Second Circuit stated in *Altai* that, in determining software copyright

issues of industrial design, e.g., the choice of one or another programming technique or the clarity (or obscurity) of the functional purpose of a portion of the program.[59]

1.4.4 The Industrial Design and Construction of Programs Rests on a Large Body of Skilled Know-How

Programmers use a substantial body of know-how in the design and implementation of programs. By "know-how" we mean "the totality of unpatented knowledge utilized in industry,"[60] which embodies " 'detailed innovation in industrial techniques' of a practical nature that is often the 'fruit of . . . experience and trial and error.' "[61]

Programmers acquire know-how in a number of different ways. Many acquire it through formal training. The curricula of computer science and related disciplines[62] formalizes know-how pertaining to the building blocks of programs. Additional formalized know-how can be found in the proceedings of the many computer-related conferences and professional journals that are published in this field.[63] Texts and journals in the field typically describe program elements at the level of algorithms and data structures,[64] rather than literal code, because the details of pro-

infringement, courts should ignore the defendant's use of efficient design elements identical to the plaintiff's. See id. at 709.

59. Designers of physical machines sometimes also exhibit elements of personal style in their work. See, e.g., Seymour M. Hersh, A Case Not Closed, The New Yorker, Nov. 1, 1993, at 80, 85–88 (discussing search for elements of personal style in the wiring work of terrorist bombers). Many small details indicative of personal style in source code, such as the naming of variables, tend to be eliminated from the program when the source code is processed by an optimizing compiler to produce object code.

60. See François Dessemontet, The Legal Protection of Know-How in the United States of America 11 (H.W. Clarke trans., 2d ed. 1976). See also sources cited in Reichman, Applied Know-How, supra note 5, at 656.

61. Reichman, Applied Know-How, supra note 5, at 656 (quoting 3 Stephen P. Ladas, Patents, Trademarks and Related Rights: National and International Protection 1617 (1975)). Know-how, of course, exists at every phase of all industrial product cycles, from conception, to design, to initial implementation, to volume manufacturing, distribution, marketing, service, and so forth. Here we focus on know-how pertinent to the design and implementation of program development because software development is "all design and no manufacture." NRC Report, supra note 4, at 44.

62. User interface design, for example, is often taught in the cognitive science programs of psychology departments.

63. Two major associations to which many computing professionals belong are the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). These associations sponsor many conferences each year on specialized subfields, such as software maintenance, user interface design, and computer graphics. They also publish numerous journals of interest to computing professionals, some of which, such as IEEE Computer and Communications of the ACM, are of a more general nature, and some of which are highly specialized, such as the ACM SIGCHI Bulletin (for designers of computer-human interactions).

64. Typical texts describe a few dozen basic data structures and several dozen basic algorithms. See, e.g., Thomas H. Cormen et al., An Introduction to Algorithms (1990) (1000 page book describing a wide variety of data structures and algorithms); Donald E.

gramming languages and particular hardware systems (which would be needed for discourse at the code level) are sufficiently complex that focusing on them would obscure, rather than clarify, know-how pertinent to program construction.[65] Specialized texts describe data structures and algorithms appropriate for particular tasks.[66] This formalized know-how is part of the expertise of a good programmer.

As in many other engineering fields, practitioners acquire additional, less formal know-how on the job. Programmers acquire much of this know-how by developing programs themselves and by working with others in teams. They also pick up informal know-how through exchanges at conferences, electronic bulletin boards, and other electronic messages.

All of this applied know-how, both formalized and tacit, forms the core knowledge at the heart of the skill of programming.

1.5 Innovation in Programs is Largely Incremental and Cumulative in Character

Innovation in software development is typically incremental.[67] Programmers commonly adopt software design elements—ideas about how to do particular things in software—by looking around for examples or

Knuth, Sorting and Searching (4th ed. 1975) (volume of multi-volume set of books on algorithms, devoted to sorting algorithms).

65. That is, the code adds detail that is relevant only to the implementation, not to the basic concept of any particular data structure or algorithm.

66. See, e.g., Andrew S. Tanenbaum, Operating Systems: Design and Implementation 213–17 (1987) (discussing page replacement algorithms, a key element in operating systems that use virtual memory).

67. See, e.g., Harlen Mills, Top Down Programming in Large Systems, *in* Debugging Techniques in Large Systems 40, 44 (R. Ruskin ed., 1971); Brooks, supra note 36, at 18. We use "innovative" to describe products of skilled effort that advance the state of the art but do not meet the patent standard of nonobviousness. Kingston uses the term in a different sense. He defines innovation as taking an invention through to commercial practice, including

> the introduction and full working of a manufacture .... Expenditure of personal effort and capital ... it was understood that the public would be instructed in the new technology by . . . [the innovator's] personal supervision of the business, which would be a tangible example, capable of being followed.
>
> The way in which this practice of exchanging monopoly for novelty developed historically, however, led to Patents for invention instead of innovation. Wherever Patents are granted today, they relate only to information, not to that information embodied, working, and tangible.

William Kingston, The Unexploited Potential of Patents, *in* Direct Protection of Innovation 1, 2 (William Kingston ed., 1987) [hereinafter Direct Protection]; see also id. at 31 ("It is of the very nature of [incremental innovation] that it evolves out of what is already there. There is a sense in which each increment in any evolutionary process is actually implicit in its predecessor, and it, in turn, points towards its own successor."); William Kingston, Advantages of Protecting Innovation Directly, *in* Direct Protection, supra, at 87, 107 ("What characterizes this type of innovation more than anything else, is that once it has been done, nothing is easier than to reconstruct it from the elements of prior art.").

remembering what worked in other programs. These elements are sometimes adopted wholesale, but often they are adapted to a new context or set of tasks.[68] In this way, programmers both contribute to and benefit from a cumulative innovation process. While innovation in program design occasionally rises to the level of invention, most often it does not.[69] Rather, it is the product of the skilled use of know-how to solve industrial design tasks.[70]

Even the conceptual metaphors embodied in software are typically incremental in character.[71] Word processing paper is an extension of traditional paper; the automated spreadsheet is an extension of the traditional paper spreadsheet. Both products are innovative, but both also derive from an existing idea: a manual version. Put more generally, software development typically involves the automation of known tasks, which, almost by definition, involves an incremental innovation. Many extensions of functionality that are built upon a program's core conceptual metaphors (such as cutting and pasting with word processing software) also involve incremental innovation, applying human factors engineering to the design task.[72]

The technical community has recognized the cumulative and incremental nature of software development, and has welcomed efforts to direct the process of software development away from the custom-crafting that typified its early stages and toward a more methodical, engineering approach.[73] The creation of a software engineering discipline reflects an

68. See, e.g., NRC Report, supra note 4, at 61.

69. See CONTU Report, supra note 17, at 17; Casey P. August & Derek K.W. Smith, Understanding Some Intricacies of Software: Expression, Interfaces, and Reverse Assembly, Computer Law., Apr. 1994, at 16, 16 ("In practice, patent protection for program ideas is only available to a very tiny fraction of computer programs, because most programs contain only widely known ideas which are merely aggregated in an obvious manner which, in the software developer's opinion, fits a particular application."); Duncan M. Davidson, Protecting Computer Software: A Comprehensive Analysis, 23 Jurimetrics J. 339, 357 (1983); Randall M. Whitmeyer, Comment, A Plea For Due Processes: Defining the Proper Scope of Patent Protection for Computer Software, 85 Nw. U. L. Rev. 1103, 1131 (1991) (expressing view that overwhelming majority of software innovations would not meet Patent Act's novelty and nonobviousness standards even if subject matter hurdles could be overcome).

70. See, e.g., Galbi, supra note 6, at 281; Reichman, Applied Know-How, supra note 5, at 658–59 (discussing skilled efforts to apply know-how in the construction of computer programs).

71. See generally Curtis, supra note 45, at 71–78 (discussing the human factors engineering that goes into user interface design).

72. See id. at 61, 73 (discussing engineering analysis that went into choice of now standard icon for representing documents).

73. See, e.g., Mary Shaw, Prospects for an Engineering Discipline of Software, IEEE Software, Nov. 1990, at 15, 20. For a general introduction to software engineering, see, e.g., B. Ratcliff et al., Software Engineering: Principles and Methods (1987); see also Barry W. Boehm, Software Engineering Economics (1981) (discussing economic reasons for an engineered approach to software development). There are over one hundred books in the University of Pittsburgh Library with the term "software engineering" in their titles.

awareness that program development requires skilled effort and applied know-how comparable to other engineering disciplines.

The products of software engineering almost invariably contain admixtures of old and new elements. Some consist almost entirely of old elements. The innovation in such programs may lie in the manner in which the known elements have been combined in a new and efficient manner. Or it may come from combining some new elements with well-known elements in order to achieve the same result in a new way. When we speak of programs as "industrial compilations of applied know-how," it is in recognition of the frequency with which software engineering involves the reuse of known elements.[74] Use of skilled efforts to construct programs brings about cumulative, incremental innovation characteristic of engineering disciplines. A well-designed program is thus akin to the work of a talented engineer whose skilled efforts in applying know-how, accumulated from years of experience and training, yields a successful design for a bridge or other useful product.

2 The Industrial Compilations of Applied Know-How in Programs Need Legal Protection that Existing Regimes Cannot Provide

This Section argues that the industrial compilations of applied know-how embodied in programs are in need of legal protection that existing regimes cannot provide. We begin in Section 2.1 by discussing some properties of software that distinguish it from typical industrial products. Chief among these is that software bears the bulk of the know-how required to make it on or near the "face" of the product distributed in the market.[75] The know-how borne in software products distributed in the marketplace makes them vulnerable to rapid, inexpensive copying that undercuts the initial developer's opportunity to benefit from the value it has contributed to the market, thereby undermining its incentives to invest in software development. We refer to this as the vulnerability of software innovations to trivial acquisition of equivalence.

Section 2.2 will show that existing legal regimes cannot effectively remedy some market-destructive appropriations of know-how borne in software because of a mismatch between fundamental principles of each regime and the loci of value in software. Because know-how borne on or

74. We recognize that the compilation metaphor has some limitations when used to refer to programs. First, programs can be composed of entirely new elements, although they rarely are. Second, there will generally be more coherence in the integration of program components than would be true for the kinds of compilations that copyright law has historically protected. See generally Jane C. Ginsburg, Creation and Commercial Value: Copyright Protection of Works of Information, 90 Colum. L. Rev. 1865, 1893 (1990) (discussing copyright protection for compilations of information). However, integration of useful components is typical of industrial design, whether of programs or of other industrial products. See Reichman, Design Protection after 1976, supra note 54; Reichman, Design Protection 1948–1976, supra note 54.

75. See Reichman, Applied Know-How, supra note 5, at 657–61 (discussing computer programs as bearing know-how on their face).

near the face of publicly distributed software products cannot be kept secret, trade secret cannot protect it. The predominantly functional nature of program behavior and other industrial design aspects of programs precludes copyright protection, while the incremental nature of innovation in software largely precludes patent protection.

Software is a valuable artifact whose know-how is largely evident in distributed products. This know-how is much cheaper to copy than to create, and is unprotected by current law, which makes innovative software developers vulnerable to appropriations that undermine their incentives to invest in software innovation. Section 2.3 shows that attempts to use existing legal regimes to protect the behavior of software and the industrial design it embodies inevitably lead to cycles of under- and overprotection that are unhealthy for the industry.

2.1 Program Innovations are Vulnerable to Trivial Acquisitions of Equivalence that Can Cause Market Distortions

2.1.1 Programs Reveal a Substantial Amount of their Know-How in Products Distributed in the Market

Know-how can inform any phase of the product development process, including design, initial construction (making one), and mass production (making many). Know-how can be found in several loci, including on the "face" of a product (i.e., evident on simple inspection), interior to the product (i.e., carried in the product but not immediately evident), and inside the factory.

Computer programs are an unusual kind of industrial product because the bulk of the know-how required to create them is accessible on or near the face of the product distributed in the marketplace. There are several reasons why this is so: (1) Software products need little specialized mass-production know-how; (2) they are particularly rich in design know-how; and (3) they are more susceptible in some respects to reverse engineering than traditional industrial products.

Software is easy to mass-produce because it is an information product. There is no steel to cut or bend, no wires to attach, and no parts to be cast in plastic or rubber. Mass production is as simple as loading copies onto floppy disks or tapes, a routine, inexpensive, and low-technology undertaking. Because there are no special processes required for mass production of the software products, there is no opportunity to accumulate specialized know-how about mass production. Lack of mass production know-how matters because this is the kind of know-how that can most easily be kept out of the product, i.e., maintained as a trade secret.[76]

76. "There is little difference between the pattern of responses for processes and for products, except that, as one would expect, reverse engineering is markedly more effective in yielding information about product technology." Richard C. Levin et al., Appropriating the Returns from Industrial Development *in* 1989 Brookings Papers on Economic Activity 783, 805–06.

Most mass-production know-how of industrial products never leaves the factory floor. Software, however, has no need for mass production technology, hence has no mass production know-how to keep secret.

Software is, instead, rich in surface design know-how. This is particularly true for interactive software, i.e., programs that involve a great deal of hands-on usage.[77] For such software, the metaphor or conception of the task is particularly important. The virtualities that programs create are crucial to organizing their complex behaviors. They provide what psychologists call an "affordance," i.e., the object itself indicates how it should be used.[78] The spreadsheet metaphor, the virtual paper in word processors, and especially the desktop metaphor all illustrate the importance and value of the design know-how that goes into creating an organizing metaphor.

Design know-how for software metaphors is especially important because software allows new conceptions of a task. Unlike physical objects, the virtual objects created in software are not constrained to obey the laws of physics.[79] Only inspiration and insight limit them, not physical or material properties. In the desktop metaphor, for example, the electronic version of file folders can expand, contract, or reorganize their contents on demand, quite unlike their physical counterparts.

Software is not only especially rich in design know-how; much of the value of software arises from it. The desktop metaphor, for example, was crucial to the early success of Apple computers and remains a major factor today.[80] Two of the best-known "look and feel" lawsuits, *Lotus Development Corp. v. Paperback International, Inc.*[81] and *Apple Computer, Inc. v. Microsoft Corp.*,[82] challenged a competitor's appropriation of interface de-

77. This includes almost all application software, such as spreadsheets, word processors, databases, graphics programs, project management programs, finance programs, and electronic mail systems, as well as many aspects of the operating system. We are less concerned here with utility programs that support the computer's infrastructure, such as memory managers or network utilities, and are not at all concerned with programs whose most important output is a single result produced by a novel process; for instance, the solution to a traveling salesman problem produced by the Karmarkar algorithm. See Andrew N. Parfomak, The The Karmarkar Algorithm—"New" Patentable Subject Matter?, 21 Int'l Rev. Indus. Prop. 31 (1990).

78. The term affordance typically refers to the appearance of a physical object, but it may also refer to icons on a computer screen or the underlying metaphor in a system. See Curtis, supra note 45, at 60–61.

79. See supra note 47.

80. Charles H. Ferguson & Charles R. Morris, Computer Wars: How the West Can Win in a Post-IBM World 106 (1993). The desktop metaphor radically changed the user's experience in using a computer, from typing obscure commands, to manipulating familiar objects. The change opened up the power of computers to many people for whom learning to use them would have been intimidating.

81. 740 F. Supp. 37 (D. Mass. 1990).

82. 799 F. Supp. 1006 (N.D. Cal. 1992), *aff'd* 35 F.3d 1435 (9th Cir. 1994).

sign, not elements of program text. There is also a good deal of commercial interest in designing software to be usable.[83]

The difficult task in creating interactive software, the innovations embodied in it, and the value of those innovations are typically found in conception and design: determining how to think about the task (e.g., the desktop metaphor), and then determining what capabilities the program should have (e.g., what can be done with documents, or folders). Determining the answers to these questions requires both informed insight and the time-consuming and expensive process of having real users test the software for extended periods.[84] We can summarize by saying that interactive software is, by its nature, rich in design know-how.

Such know-how is, inescapably, present on the face of the product, and immediately evident on inspection. The hard-won insights and innovations embodied in surface design are prominently displayed by the program in operation; they are also explained in online help, and further detailed in the manual. Insights not readily explained by the tool or the manual will often become evident to a sophisticated user of the program who runs the program over a variety of examples. Since program innovation often lies in finding a way to make program behavior easier for people to use, the behavior and its insights must be evident to the user.[85] As one example, Paperback needed nothing more than the ability to run Lotus 1-2-3 and observe its behavior in order to clone it. And in one of the now-legendary events of the computer industry, Steve Jobs visited Xerox's Palo Alto Research Center in 1979 for a tour that included a demonstration of Xerox's Alto computer and its graphical user interface. Many of the ideas he saw there later appeared in the Apple Lisa and Macintosh computers.[86] Hence we say that programs bear a considerable amount of know-how on the face of the product.[87]

Although much of the know-how embodied in programs is borne on the face of software products, other know-how resides in the details of its internal construction, such as in its algorithms, data structures, and con-

83. See Curtis, supra note 45, at 59; Scott Leibs, Why Can't PCs Be More Fun, Information Week, Aug. 15, 1994, at 27 ("[I]nterface design is hot. What would a truly easy—and more productive—interface look like?").

84. See NRC Report, supra note 4, at 56 (" 'The original version of [Lotus] 1-2-3 and its interface required a great deal of hard work—hundreds of hours, multiple iterations,' explained [Mitchell] Kapor, co-developer of the . . . program. 'It was very non-obvious, and it was tested on users.' "); see also Werner L. Frank, Critical Issues in Software 22 (1983) (estimating that only about 20% of software development work is attributable to writing code). The proportion of software development attributable to design, coding, testing, and maintenance may, however, vary substantially from program to program.

85. See, e.g., Alfred V. Aho et al., Data Structures and Algorithms (1983); Richard E. Fairley, Software Engineering Concepts (1985).

86. The head of the Xerox Lab subsequently recalled: "To allow Jobs to see the power of the [Alto] system . . . was a dumb thing to do . . . . Once he saw it, the damage was done; he just had to know that it was doable." Douglas K. Smith & Robert C. Alexander, Fumbling the Future 241–42 (1988).

87. See Reichman, Applied Know-How, supra note 5, at 659–60.

trol structures.[88] The innovation in program internals often lies in constructing new ways to organize and structure these information components. Program internals are an important part of the value in software because they contribute to innovative and efficient (and consequently, valuable) behavior.

One way to gain access to the internal design of a program is to get a license from its developer. A more difficult, but nonetheless workable, way to gain access to program internals is to decompile it.[89] The object code in software packages can, through a decompilation process, be translated into a form that approximates to some degree the source code that the program's developer maintains as a trade secret. While decompiled code is difficult to read, it is far easier to understand than object code.[90] It is not uncommon for software developers to decompile other firms' programs when they need access to specific crucial pieces of know-how embodied in the object code.[91]

Software developers do not use decompilation more frequently because in the current state of the art, decompilation is a painstaking and time-consuming process. Thus, although internal know-how is accessible from an examination of the software as distributed in the marketplace, it is today discernible only by the difficult process of decompilation. Hence, we speak of the internal know-how of software as lying "near the surface" of the product. As the technology for the reverse engineering of

88. See supra note 37 for a definition of algorithms and data structures. Control structures are "the facilities of a programming language that specify a departure from the normal sequential execution of statements." Websters' New World Dictionary of Computer Terms, supra note 37. See generally Davis, Nature of Software, supra note 5, at 317–25 (explaining and giving concrete examples of various kinds of program elements, including control structures).

89. The standard process of translating from source code to object code is referred to as compiling; hence the inverse process is known as decompiling. For programs that are distributed in assembly form, disassembly is a similar reverse analysis process. See Finding a Balance, supra note 2, at 7, 147–50 (nontechnical description of decompilation and disassembly).

90. Decompiled code is not as useful as source code. Decompiling cannot, for example, restore the mnemonic names for variables and procedures chosen by the programmer. Programmers select those names to help them keep track of their own code, by making clear in the name what each piece of code is doing. Hence they greatly assist anyone else who wishes to understand the code. But those names are lost in the translation from source to object code and cannot be recreated during reverse analysis.

91. The videogame developer, Atari Games, once decompiled a small program embedded in Nintendo videogame consoles. It did so to develop a program for its games that would send the precise "message" needed for them to operate in the Nintendo console. By this decompilation, Atari gained access to crucial internal know-how in the Nintendo code, know-how that had previously prevented unlicensed game cartridges from running on Nintendo consoles. After Atari used this knowledge to construct and sell videogame cartridges that would operate in Nintendo consoles, Nintendo sued, alleging copyright infringement. See Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832, 843–45 (Fed. Cir. 1992) (holding that decompilation did not infringe copyright, but affirming grant of preliminary injunction because of similarities in programs arising from Atari's efforts to achieve more than present compatibility with Nintendo systems).

programs improves, the internal know-how of programs may come to lie ever nearer the surface of the product.

2.1.2 Software Products Are Vulnerable to Trivial Acquisitions of Behavioral Equivalence

Because software bears the bulk of its know-how on or near the surface of the product, and because it is an information product, it is more vulnerable than traditional industrial products have generally been to trivial acquisition of behavioral equivalence.[92] By trivial acquisition, we mean that producing an imitation requires only modest effort, time, and cost by comparison to the resources required to develop the product initially. By behavioral equivalence, we mean that a second comer can produce a program that is indistinguishable to users, and hence, can be a market substitute for the first.[93]

There are a number of ways by which second comers can acquire behavioral equivalence with a particular program.[94] The most trivial way to acquire it is by literal duplication of the program's source or object code.[95] The next most trivial means of acquiring behavioral equivalence is by working from the other programmer's flowchart.[96] A somewhat

92. Any product with know-how on its face suffers to some extent from the problem of nonexcludability: the revealed know-how becomes a public good, i.e., a valuable entity whose benefit is freely available to all. Producers of public goods may be unable to capture a sufficient share of the benefit from such goods to justify the investment necessary to create them, leading to underproduction. See Menell, supra note 1, at 1059 (discussing public goods and concomitant market failure). One traditional economic justification for intellectual property rights is the goal of avoiding this market failure. It has been recognized for some time that innovations of the sort found in software are classic examples of public goods. See id. We suggest that software innovation has a particularly compelling public-goods problem and may need legal protection because most of its valuable know-how is on the face.

Legal protection is, however, neither guaranteed to solve the problem, nor is it the only possible solution. Other factors, notably lead time, can help considerably. See id. at 1060. Software, however, has little natural lead time.

93. See supra note 28 for factors that may make clone products less than perfect market substitutes.

94. Trivial acquisitions of equivalence can occur with other electronic information products as well. In the case of databases, for example, acquisition of equivalence can also be accomplished by exhaustive inquiry. The process can be made trivial by automating it: directing one program to send an exhaustive set of inquiries and then building its own database from the replies.

95. Copyright law addresses this kind of market-destructive copying. See, e.g., Apple Computer, Inc. v. Formula Int'l, Inc., 725 F.2d 521 (9th Cir. 1984).

96. This method does not involve access to the first program or copying of the graphical elements of the flowchart. The industrial design of the program's functional components depicted in the flowchart would no more be protected by copyright law than the industrial design of a bridge would be protected by the copyright in an engineering drawing of it. See, e.g., Kern River Gas Transmission Co. v. Coastal Corp., 899 F.2d 1458 (5th Cir.), (route for natural gas pipeline not protectable by copyright law), cert. denied, 111 S.Ct. 374 (1990); Fulmer v. United States, 103 F. Supp. 1021 (Ct. Cl. 1952) (parachute design not protected by copyright in drawing); Muller v. Triborough Bridge Auth., 43 F.

more demanding means of acquiring behavioral equivalence is by carefully studying a program's behavior by running the object code with a variety of inputs, then reproducing that behavior with newly written source code. Even so, this effort is generally trivial by comparison with the original developer's costs. The most arduous way to acquire behavioral equivalence with another program is by decompilation. This method is presently practical only for very modest amounts of code.[97]

Any product that bears a large quantum of its know-how on its face is vulnerable to rapid imitative copying because this know-how cannot be kept secret. Creating, testing, and revising the menu tree of Lotus 1-2-3, the desktop metaphor of the Xerox Alto, and the properties of the virtual paper in the first word processors all required extensive time, effort, and expense. Yet once designed, they were trivially recreated by second comers, vastly reducing lead time.[98]

Because software is an information product, it is vulnerable to trivial acquisition of production and distribution equivalence. As we have noted, the mass production of software is technologically trivial and essentially error free. Accordingly, a second comer faces no significant delays to accumulate product stock, no capital cost or delay to build or retool factories or to develop mass production processes, and no issues of production quality (error rates in conventional methods of duplicating disks and tapes are very low). The recent appearance of network distribution suggests that product distribution may become almost free and instantaneous as well.[99]

Loss of lead time is of particular concern in industries that sell information artifacts, such as software, because there are virtually no manufacturing costs associated with information artifacts in electronic form. Information is what the product has to offer, and it is there for the taking. By comparison, in industries producing physical products, some natural lead time generally exists by virtue of the need to obtain the necessary raw materials, retool one's manufacturing facility to produce a new or

Supp. 298 (S.D.N.Y. 1942) (design for traffic approach to bridge not protectable by copyright in drawing). See infra notes 179–181 and accompanying text for a discussion showing that this principle of United States copyright law was intended to apply to the functional design for programs depicted in flowcharts.

97. See supra notes 89–91 and accompanying text.

98. See NRC Report, supra note 4, at 2. See also Levin et al., supra note 76, at 809 (indicating that 84% of industry managers surveyed believed they could replicate a typical unpatented new product for less than three-quarters of innovator's research and development cost, while 53% believed they could do it for less than half of innovator's cost).

99. Software has recently begun to be delivered via networks (such as via Internet). If network distribution becomes more common, it will eliminate the time and expense of copying, packaging, and distributing disks. This kind of information product could thus (after the first artifact has been created) be manufactured, reproduced, and distributed at virtually zero marginal cost.

revised product, and distribute the manufactured items.[100] Manufacturers of standard industrial products also tend to have lead time by virtue of the trade secrets that can successfully be kept in the plant because the product does not reveal crucial know-how on or near its surface.

2.1.3 Trivial Acquisition of Equivalence Can Cause Market Failure

We stress the triviality of some acquisitions of behavioral equivalence because this triviality can be the source of market-destructive effects.[101] The particular means a firm uses to acquire behavioral equivalence is not important from a market-preservation standpoint. The crucial concern is whether it is trivially easy and quick to copy a software innovation that was very expensive to develop. If the disproportion in cost of copying and cost of innovation is substantial enough, it can destroy the innovator's opportunity to recoup its expenses, and consequently, can destroy incentives to invest in software innovation.[102]

Professor Gordon has noted that market failure of the sort under discussion requires the confluence of several distinct conditions:[103]

100. Of course, natural lead time can sometimes be quite brief for industrial products as well. This is the principal reason that some countries have adopted utility model and industrial design laws to give artificial lead time to incremental innovation in industrial products. See J.H. Reichman, Legal Hybrids Between the Patent and Copyright Paradigms, 94 Colum. L. Rev. 2432, 2455-62 (1994) [hereinafter Reichman, Legal Hybrids].

101. Copying that is trivially easy and rapidly accomplished will not create market distortions if the costs of the initial development are about the same as the cost of copying or if there are obstacles that prevent one who has engaged in rapid and easy copying from reaping substantial rewards from this practice (e.g., if the consuming public did not trust a clone product). Market destructive effects will occur when the costs of initial development are high, the costs of copying are very low, and the copyist can undercut the innovator's price and obtain substantial profits by taking a free ride on the innovator's efforts. In such a situation, firms may underinvest in innovation. See, e.g., Wendy J. Gordon, Assymetric Market Failure and Prisoner's Dilemma in Intellectual Property, 17 U. Dayton L. Rev. 853, 861-67 (1992) [hereinafter Gordon, Market Failure]; see also Wendy J. Gordon, On Owning Information: Intellectual Property and the Restitutionary Impulse, 78 Va. L. Rev. 149, 222-58 (1992) (discussing a proposed tort of "malcompetitive copying").

102. See, e.g., Dreyfuss, supra note 6, at 901-02 (contrasting the lead time generally available to developers of traditional products with that available to developers of information products). Underinvestment in software innovation will, in turn, lead to underproduction of follow-on generations of goods and services that would have been produced if the initial goods had been available in the market. See, e.g., Gordon, Market Failure, supra note 101, at 854.

103. Gordon, Market Failure, supra note 101, at 863-65. Professor Gordon has employed a variant on the "prisoner's dilemma" problem from decision theory to show that the grant of an intellectual property right can sometimes avoid this kind of market failure. In the standard prisoner's dilemma model, two persons charged with a crime face a dilemma: whether to decline or accept an offer of a lighter sentence in exchange for testimony against a fellow suspect. If both stay silent, each will receive a sentence of, say, 18 months; if both accept the offer, both will be sentenced to, say, five years in prison; if only one accepts the offer, that person will go free and the other will do a maximum prison term of, say, nine years. Although both would be better off if both declined the

1. the costs of creating an innovative product are high;
2. the costs of copying are very low;
3. the products offered by the copyist are equivalent to those offered by the innovator;[104]
4. consumers know the two products are equivalent;
5. the investment in innovation would be more than paid off if no copying occurs;
6. the investment will be lost if copying occurs because the copier is able to save so much by copying that he can charge a lower price than the innovator.[105]

Gordon posits two potential innovators, considering an investment in innovation, but uncertain of whether the other will innovate or imitate. In such a situation, she says, the optimal outcome, both for the parties and for society, is for both to choose to invest in development of an innovative product.[106] However, when both fear the other will yield to the temptation to copy, both are likely to opt not to innovate.[107] A legal proscription against copying can prevent these actors from engaging in mutually destructive conduct (not innovating) and would cure the market failure that unconstrained copying would cause.[108]

These conditions can often be met in the software market. As we have observed, the costs of software development are high; the costs of copying can be extremely to somewhat trivial; copying can produce an equivalent product that will be recognized as such by consumers; investments can be recouped if copying is restrained; and firms that sell clones of software products can sell lower-priced products because of the disparity between the costs of the innovator's development and the costs of copying. The innovator's investment can be lost if consumers succumb to the temptation to buy the lower-priced product.

Focusing on trivial acquisition of functional equivalence directs discussion to issues of importance to software developers—the source of value in software (know-how pertaining to behavior and the design that brings it about) and the results of copying (achieving behavioral equivalence)—rather than to issues dictated by traditional legal regimes, such as the medium of creation (program code vs. a hardware equivalent), or to concepts such as idea and expression, whose distinction is extremely

prosecution's offer, the common outcome of this dilemma is that both accept the offer and do more time than if both had declined. See id. at 861. The payoff structure of these choices can, however, be adapted to other contexts. In intellectual property terms, the cooperative strategy is to be creative, and the defection strategy is to copy. See id. at 863.

104. Equivalence may also be needed with respect to factors affecting consumer satisfaction (e.g., service, reliability, maintenance, upgrades, number of distribution outlets, etc.).

105. Gordon, Market Failure, supra note 101, at 863.

106. See id. at 864.

107. See id.

108. See id. at 864–65.

difficult to make clear and operational in relation to software.[109] In addition, it directs discussion to the nature of the results to be avoided (trivial acquisition of behavioral equivalence) rather than to any particular doctrinally significant means by which to achieve behavioral equivalence (such as decompilation).[110]

2.1.4 Trivial Acquisition of Behavioral Equivalence Is a Technology-Independent Concept

Focusing on trivial acquisition of behavioral equivalence offers a degree of independence from current technology. This is important for two reasons: first, in the future, new ways may be found to acquire behavioral equivalence with another program;[111] second, the market-destructive potential of some existing techniques for achieving equivalence may be altered by the evolution of technology. The latter point can be illustrated by considering the current and potential future practice of decompilation.

Reverse engineering of the contents of a program by decompilation or disassembly is currently a largely manual and very tedious process involving considerable effort to learn anything of value about a program of more than modest size.[112] Given today's technology, programmers generally cannot hope to use decompilation as a means to achieve behavioral equivalence with only trivial effort. Because of this, under a trivial acquisition test, current industry "clean-room" reverse-engineering processes would be lawful ways to acquire behavioral equivalence.[113]

But as automated tools for reverse engineering improve,[114] as they are expected to do,[115] the day may come when reverse-engineering tech-

109. See infra notes 197–209 and accompanying text for a discussion of the difficulties courts have encountered in applying the idea/expression distinction to software.

110. See infra notes 325–337 and accompanying text for a discussion of the doctrinal controversy about decompilation and the extent to which the doctrinal discussion obscures, rather than clarifies, the true significance of decompilation.

111. See, e.g., Brooks, supra note 36, at 15–16 (discussing the prospects for automatic and graphical programming).

112. See, e.g., Finding A Balance, supra note 2, at 147–48.

113. See supra note 24.

114. We emphasize here the potential impact of improvements in reverse-engineering technology because they could make program internals more transparent and hence easier to appropriate. However, any technique that improves the efficiency of constructing software can affect the ease and rapidity with which software innovations can be appropriated.

115. See NRC Report, supra note 4, at 11, 78. It is surely true that reverse-engineering technology will improve. See generally Jim Q. Ning et al., Automated Support for Legacy Code Understanding, Comm. ACM, May 1994, at 50 (discussing program segmentation as advanced form of reverse engineering). The larger and more complex the program, the greater will be the level of difficulty for a copyist to find any particular capability in the object code and decompile it. Furthermore, increased use of more powerful optimizing compilers may make decompilation more difficult over time rather than less. Optimizing

nologies would permit a firm to create a functional equivalent of a program with only trivial investments of time, effort, and expense. In that case, use of these techniques on someone else's software without permission might need to be regulated. Those who have tried to persuade courts and policymakers to outlaw decompilation[116] may realize that a future in which decompilation can be done with trivial effort may not be so comfortably far off as to be mere conjecture.

2.2 Existing Legal Regimes Cannot Appropriately Protect Industrial Design Elements of Programs

This subsection will show that existing legal regimes are structurally unsuited to providing an appropriate degree of protection to compiled know-how lying on or near the face of software products. Trade secrecy law cannot protect know-how borne in products found in the marketplace. Because programs are both literary works and machines, both copyright and patent law have had difficulties dealing with them.[117] In addition, the independence of program text and behavior make text-oriented solutions to protecting behavior unworkable.

2.2.1 Trade Secrecy Law Cannot Protect Know-How Discernible in Publicly Distributed Products

The mismatch between existing legal regimes and the know-how embodied in software is most evident with respect to trade secrecy law. Although trade secrecy law has a long history of protecting industrial compilations of applied know-how,[118] it cannot protect behavior or other

compilers are high-performance compilers capable of producing object code that executes especially quickly. This execution speed is, however, typically bought at the cost of making the object code considerably more difficult to reverse analyze, even for the author of the source code. Hence, a second comer intent on decompiling the object code may find that the size of the program and the complexities introduced by a high performance compiler will more than offset the added power of more advanced reverse-engineering tools. The balance of power in these two technologies will bear watching as it evolves. Our thanks to Professor Lee Hollaar for sharing his thoughts on this issue.

116. See infra note 329 and accompanying text.

117. See, e.g., Computer Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 712 (2d Cir. 1992) (referring to "hybrid nature" of computer programs and indicating that utilitarian nature of programs made it difficult to apply idea/expression distinction); OTA Report, supra note 6, at 78–80. Regarding patent law difficulties with programs, see infra notes 131–140 and accompanying text. Theodor Nelson coined the term "literary machines" in a book about a hypertext publishing system. See Theodor H. Nelson, Literary Machines (5th ed. 1983). We think the term has a more general application to software.

118. See Restatement of Torts § 757 cmt. b (1939) (defining subject matter of trade secrecy law as including compilations of information used in industry to give firm a competitive advantage over those who do not possess them). For a recent case in which compiled know-how was treated as a trade secret, see Rockwell Graphic Sys., Inc. v. DEV Indus., 925 F.2d 174, 176–77 (7th Cir. 1991) (treating compiled know-how about materials, dimensions, tolerances, and methods of manufacture as trade secrets). For in-depth coverage of American trade secrecy law, see Roger M. Milgrim, Milgrim on Trade Secrets

know-how borne on the face of a mass-marketed software product because such know-how cannot be kept a secret. An experienced programmer who runs a program to study its component behaviors can often learn everything necessary to make a functionally indistinguishable program.

Nor can trade secrecy law protect know-how borne near the surface of software products, because trade secrecy law has long regarded reverse engineering of products available in the marketplace as a fair means of acquiring trade secrets.[119] Thus, a competitor can buy an industrial product in the marketplace, disassemble it to discern how and of what the product is made, and make a similar product. As applied to software, this policy would suggest that reverse analysis of program code by decompilation or disassembly would be a fair means of obtaining the compiled know-how borne near the surface of software products.[120]

2.2.2 Why Patent Law Is Ill-Suited to Protecting Software Innovation

The dual character of computer programs—which are both writings and machines at the same time—has presented some difficulties for those wanting to use patent law as a means of legal protection for software innovations. Patent law has traditionally excluded texts ("printed matter"), as well as methods embodied in texts, from its do-

(1994). For economic analyses of trade secrecy law, see David D. Friedman et al., Some Economics of Trade Secret Law, 4 J. Econ. Persp., Winter 1991, at 61; Edmund W. Kitch, The Law and Economics of Rights in Valuable Information, 9 J. Legal Stud. 683, 708–23 (1980).

Although secret know-how has received legal protection for centuries, the law of trade secrecy, as such, is of more recent origin. Until the mid-1800s, the tort doctrine on breaches of confidential relationships and the law of contracts were the legal doctrines employed to protect commercially valuable secrets. The case that seems to have given rise to a separate law of trade secrecy was Morison v. Moat, 9 Hare 241, 68 Eng. Rep. 492 (V.C. 1851) (enjoining defendant from selling version of unpatented medicine because of wrongful manner of acquisition from originator). The misappropriator of commercially valuable information in that case had neither a contractual nor a confidential relationship to the plaintiff, yet was clearly on notice that the plaintiff had kept certain information secret in order to protect its commercial advantage in the marketplace. See id. at 495–98. Although the idea of trade secrecy law caught on in the United States, there are some countries that still do not have a separate law of trade secrecy. See Finding a Balance, supra note 2, at 88.

119. See, e.g., Kewanee Oil Co. v. Bicron Corp., 416 U.S. 470, 476 (1974) (construing Ohio statutes and Restatement of Torts § 757). If the defendant takes the market-destructive approach of appropriating the information from material given to it in confidence by the plaintiff, rather than actually doing the reverse engineering, courts sometimes find trade secret misappropriation. See, e.g., Smith v. Dravo Corp., 203 F.2d 369, 374–75 (7th Cir. 1953) (holding argument of wrongful acquisition not estopped by possibility of legal acquisition).

120. See infra note 329 and accompanying text for a discussion of the efforts some have made to prohibit decompilation, as a matter of copyright law, so that the contents of program texts can be protected as trade secrets.

main.[121] Under this doctrine, program code is unpatentable even though, once in machine readable form, it is unquestionably a technological process.[122]

Under the "printed matter" rule and the kindred "mental process" doctrine,[123] more abstract elements of programs should also be unpatentable.[124] Perhaps out of deference to their traditionally separate domains, patent law has generally left innovations in the representation, ordering, and presentation of information to copyright.[125] The Patent and Trademark Office and the courts have historically explained that such innovations lacked an industrial character.[126]

In reliance on these doctrines, the Patent Office for nearly two decades, refused patent applications for software innovations on subject

121. See, e.g., In re Rice, 132 F.2d 140, 141 (C.C.P.A. 1942) (holding that a pictorial method of writing music was not patentable subject matter under the "printed matter" rule); In re Russell, 48 F.2d 668, 669 (C.C.P.A. 1931) (holding method of arranging directories in phonetic order unpatentable). See generally Note, The Patentability of Printed Matter: Critique and Proposal, 18 Geo. Wash. L. Rev. 475 (1950) (exploring case law development of "printed matter" doctrine). The Patent and Trademark Office currently relies on the "printed matter" rule as its basis for excluding programs from the subject matter of patent law. See, e.g., United States Patent and Trademark Office & United States Copyright Office, Patent-Copyright Laws Overlap Study 46 (1991) [hereinafter Overlap Study].

122. It is because programs in machine-readable form are utilitarian processes that Professor Chisum has argued that program instructions should be regarded as patentable subject matter. See 1 Donald S. Chisum, Patents § 1.02[4] (1994).

123. A number of appellate courts have ruled that information processes are unpatentable. See, e.g., In re Abrams, 188 F.2d 165, 168 (C.C.P.A. 1951) (holding that method involving "calculating," "measuring," "observing," and "comparing" of data elements was unpatentable). The Patent Office has regarded information processes to be unpatentable even if an applicant claims more than a mental implementation of them (e.g., use of some technology, such as pen or calculator, to carry them out). See Samuelson, *Benson* Revisited, supra note 5, at 1043–44. The United States Supreme Court has stated that "mental processes" are unpatentable subject matter. See Gottschalk v. Benson, 409 U.S. 63, 67 (1972); see also In re Meyer, 688 F.2d 789, 795–96 (C.C.P.A. 1982) (rejecting claims for programmable medical diagnosis system as merely simulating thought processes of doctors). The name generally given to the doctrine established in this line of cases is the "mental step" or "mental process" doctrine. See 1 Chisum, supra note 122, § 1.03[6].

124. Mathematical principles, like laws of nature and other scientific principles, have also been regarded as unpatentable in character. See, e.g., *Gottschalk*, 409 U.S. at 71–72 (program algorithm held unpatentable mathematical principle).

125. See generally Ginsburg, supra note 74 (tracing development of copyright protection for selection, arrangement, and manner of presentation of information).

126. Both relied on Cochrane v. Deener, 94 U.S. 780 (1877), as having established that patentable "processes" were those that involved the transformation of matter from one physical state to another. See *Gottschalk*, 409 U.S. at 69–70; see also Samuelson, *Benson* Revisited, supra note 5, at 1033–40, 1056–57 (discussing cases and reasons for this interpretation of process). For a discussion of some processes that have long been thought to be ineligible for patent protection, see 1 Chisum, supra note 122, § 1.03. But see Chisum, Algorithms, supra note 1, at 1020 (arguing that program algorithms should be patented).

matter grounds.[127] The U.S. Supreme Court seemed to concur in this legal position, and its decisions influenced the patent policies of other nations.[128] Since the mid-1980s,[129] however, patent protection has become available for a wide range of software innovations, seemingly because of the perceived need for more legal protection of programs than copyright alone could provide.[130]

Apart from subject matter concerns, one reason that patents have limited application in the protection of behavior is that patents typically issue for particular methods of achieving results, rather than for results themselves.[131] It is quite possible to produce functionally indistinguishable program behaviors through use of more than one method.[132] This means that holding a patent on one method of generating certain results could not prevent the use of another method, even if those results were the program's principal source of value.[133] Hence, patents on methods would not protect behavior and therefore would not protect the primary entity of value in software. Yet if the Patent and Trademark Office were willing to issue a patent with claims for *any* means of achieving a particular set of results, such a patent would issue at a high level of generality and would inhibit competition in development of useful program behaviors out of proportion to the innovation actually contributed by the claimant.[134]

---

127. The Court of Customs and Patent Appeals overturned some of these subject matter rulings. See Samuelson, *Benson* Revisited, supra note 5, at 1041–48.

128. See Diamond v. Diehr, 450 U.S. 175, 185–87 (1981) (holding rubber curing process that included computer program patentable, but reaffirming *Benson* and *Flook* on unpatentability of algorithms); Parker v. Flook, 437 U.S. 584 (1978) (holding algorithm unpatentable, even though industrial application specified); *Gottschalk*, 409 U.S. at 63 (holding algorithm unpatentable). For international influences, see Henri Hanneman, The Patentability of Computer Software 243 (1985) (discussing impact on European patent policy for software innovations); see also Samuelson, *Benson* Revisited, supra note 5, at 1132–33 & n.426 (discussing standards for patenting software-related innovations in Europe).

129. See Samuelson, *Benson* Revisited, supra note 5, at 1093–94.

130. See Barton, supra note 6, at 265 (suggesting this rationale for the revised patent policy).

131. See, e.g., U.S. Patent No. 4,821,211 to Torres, issued Apr. 11, 1989 (Method of Navigating Among Program Menus Using a Graphical Menu Tree); U.S. Patent No. 4, 796, 220 to Wolfe, issued Jan. 3, 1989 (Method of Controlling the Copying of Software).

132. See, e.g., 3 Knuth, supra note 64 (volume describing various ways to accomplish sorting and searching).

133. Some holders of software-related patents have, however, tried to claim all means of achieving particular results. See, e.g., Brian Kahin, The Software Patent Crisis, Tech. Rev., Apr. 1990, at 53, 58 (giving examples of overbroad patents for performing certain functions by computer).

134. See, e.g., U.S. Patent No. 5,317,757 to Medicke & Posharow, issued May 31, 1994 (System and Method for Finite State Machine Processing Using Action Vectors); U.S. Patent No. 5,105,184 to Pirani & Ekedal, issued Apr. 14, 1992 (Methods for Displaying and Integrating Commercial Advertisements with Computer Software).

Other factors thought by some to impede the utility of patents in the software industry include the high costs and long delays required to obtain such protection for program

However, the more profound problem with using patent law to protect functional program behavior, user interfaces, and the industrial design of programs that produce behavior is that these innovations are typically of an incremental sort.[135] Classical intellectual property regimes do not protect this kind of innovation.[136] Patent law requires an inventive advance over the prior art before it grants protection.[137] Protecting incremental innovations in program behavior through patent law would thwart the economic goals of the patent system: to grant exclusive rights only when an innovator has made a substantial contribution to the art and advanced competition to a new level.[138]

In this respect, software resembles semiconductor chips whose industrial designs are rarely inventive. Congress recognized that the chip industry's products were both patentable and vulnerable to rapid imitative copying that undermined innovators' ability to recoup research and development costs; this undermined incentives to make the substantial investments necessary to develop new chip designs.[139] To provide proper incentives for semiconductor designs, Congress passed the Semiconductor Chip Protection Act of 1984 (SCPA).[140] It may eventually need to do the same for computer software, where the typically incremental nature of innovation also impedes the utility of patent protection.

In addition, the application of both copyright and patent law to software innovations may impair the effectiveness of both forms of protection. It has also created considerable uncertainty about the scope of pro-

innovations. See, e.g., Garfinkel et al., supra note 5, at 52; Kahin, supra note 133, at 55. As a practical matter, these barriers might make patent protection unavailable to small software developers with limited resources. Software developers generally need legal protection most in the first few years after they introduce an innovation to the market, a need which patents cannot meet. United States patent law provides no protection to an invention (except if it can be kept as a trade secret) until the patent actually issues. This rule makes many software innovations vulnerable to appropriation in the first years after they are introduced into the market. If and when a patent finally issues, it may also come after the useful commercial life of the product embodying the innovation, because of the fast pace of innovation in the industry. A form of legal protection for the first years of a technical innovation in software would, accordingly, be better tailored to the needs of software entrepreneurs. See infra text accompanying notes 411, 414.

135. See supra notes 67–74 and accompanying text.

136. Trade secrecy law protects incremental innovations that can be kept a secret. See Reichman, Legal Hybrids, supra note 100 (concerning legal hybrid regimes, created to protect some publicly disclosed incremental innovations).

137. See 35 U.S.C. § 101 (1988).

138. See U.S. Const. art. I, § 8, cl. 8; Michael Lehmann, The Theory of Property Rights and the Protection of Intellectual and Industrial Property, 16 I.I.C. (1985).

139. See H.R. Rep. No. 781, 98th Cong., 2d Sess. 1–40, (1984), reprinted in U.S.C.C.A.N. 5750, 5750–60.

140. 17 U.S.C. §§ 901–914 (1988). SCPA is discussed further infra notes 379–381, 417–428 and accompanying text.

tection available from each.[141] No one knows just where the boundary line between these domains does or should lie.[142] The economic goals of both regimes can be thwarted when both are applied to a dual-character subject matter such as computer programs.[143] This is especially true if copyright law, with its long duration of protection, its low creativity threshold, and its automatic protection, is construed so broadly that it encompasses technical innovations that should be regulated by patent law.[144]

2.2.3 Why Copyright Law Is Ill-Suited to Protecting Software Innovations

The dual nature of programs has also created conceptual difficulties for copyright law.[145] For a time, copyright officials were unsure that computer programs in machine-readable form were copyrightable subject

141. See Pamela Samuelson, Survey on the Patent/Copyright Interface for Computer Programs, 17 AIPLA Q.J. 256, 259-61 (1989) (showing differences of opinion among computer lawyers).

142. Is a program's data structure copyrightable, patentable, both, or neither? For discussion of this issue under each law without regard to the other, see Whelan Assocs. v. Jaslow Dental Lab., Inc., 797 F.2d 1222 (3d Cir. 1986) (seeming to regard data structures as copyrightable), cert. denied, 479 U.S. 1031 (1987); In re Bradley, 600 F.2d 807 (C.C.P.A. 1979) (data structure for microcode function held patentable), aff'd sub nom. Diamond v. Bradley, 450 U.S. 381 (1981) (equally divided court); see also In re Lowry, 32 F.3d 1579 (Fed. Cir. 1994) (holding data structure patentable subject matter); In re Warmerdam, 3 U.S.P.Q.2d (BNA) 1754 (Fed. Cir. 1994) (holding data structure unpatentable, although claim for machine implementation of data structure was patentable).

143. See Reichman, Legal Hybrids, supra note 100, at 2451-52 (discussing the economic goals of the classical intellectual property regimes).

144. See OTA Report, supra note 6, at 83. This problem is especially serious in light of the interchangeability of software and hardware. See supra note 27 and accompanying text. At the moment it is generally not feasible to transform hardware into software embodiments. If the technology develops to make this transition trivial, however, the inventor of any piece of digital electronics could acquire 75 years of protection without having to prove nonobviousness or to pass an examination of the prior art. It would be odd if the choice between two interchangeable media in which a given innovation could be embodied, rather than the innovation itself, determined the form of legal protection available for it.

145. A senior patent attorney of IBM Corp. was one of the first to perceive the dual nature of programs as texts and machines. See Galbi, supra note 6, at 281. Programs are not the first kind of work to have a dual character. Architectural works, for example, have a dual character because they may be aesthetically pleasing and functional at the same time. Literary works, however, have rarely had a dual character. The closest thing to a discussion of dual-natured literary works in the copyright literature is the case of Baker v. Selden, 101 U.S. 99 (1879). In *Baker*, the Court ruled that sample ledger sheets that accompanied a book explaining an accounting system were unprotectable by copyright law because they were functional, constituent parts of the accounting system described in the book. See id. at 104-05. In the United States, and in many other countries (the most notable exception being France), dual-natured industrial designs, such as those of furniture, kitchen utensils, and automobiles, are generally unprotectable by copyright law. See Reichman, Legal Hybrids, supra note 100, at 2461. Some countries (although not the United States) have special laws to protect dual-natured industrial designs. See id.

matter.[146] Copyright law has traditionally excluded machines and technological processes from its domain, leaving their protection to patent law.[147]

146. It is worth noting that the United States Copyright Office initially had doubts about the copyrightability of machine-executable code. The Office recognized that the functionality of programs in machine-executable form might disqualify them from copyright protection under principles from the Supreme Court decisions in White-Smith Music Publishing Co. v. Apollo Co., 209 U.S. 1, 17 (1908) (concerning unreadability) and *Baker*, 101 U.S. at 104–05 (concerning functionality). See George D. Cary, Copyright Registration and Computer Programs, 11 Bull. Copyright Soc'y 362, 364–65 (1964). It decided, however, in the mid-1960s, to issue registration certificates for programs, although the certificates bore witness to the Office's doubts as to the validity of claims to copyrights in machine-executable programs. See id. Relatively few registrations occurred in the decade and a half after this policy commenced. See CONTU Report, supra note 17, at 29–30.

During the 1970s, proposals for sui generis protection for computer programs emanated from a number of sources, including the World Intellectual Property Organization and the Japanese Ministry of Trade and Industry. See WIPO Proposal, supra note 6, at 6–9; Karjala, supra note 6, at 54–55. These proposals attempted to tailor protection to the unusual nature of programs. Indeed, for a time, it seemed as though the international consensus would favor sui generis protection, although this is easy to forget in light of subsequent developments. See, e.g., Dworkin, supra note 6.

The turning point in the international debate about legal protection for computer programs came in 1980 when the United States Congress endorsed the recommendations of the CONTU Commission favoring copyright protection for programs. See Act of Dec. 12, 1980, Pub. L. No. 96-517, 94 Stat. 3015, 3028 (amending 17 U.S.C. § 117); CONTU Report, supra note 17, at 23. Once the United States became firmly committed to copyright protection for programs, it began a campaign to persuade other countries to adopt the same approach because the export markets for its software products (which, then as now, dominated the world software market) depended on effective legal protection on an international basis for this very easily copied product. The first substantial victory in the United States' international campaign for copyright was in Japan. See, e.g., Barton, supra note 6, at 266. Even after this victory, European policymakers and intellectual property professionals continued to debate whether to adopt copyright or sui generis protection for computer programs. In 1991 the European Council settled on copyright, albeit with some sui generis provisions, in its Directive on the Legal Protection of Computer Programs. See EC Directive, supra note 7. This settled the international debate about the use of copyright to protect computer program code. See Dworkin, supra note 6. The recently adopted GATT/TRIPs includes a provision requiring member states to protect computer programs by copyright law. See GATT/TRIPs, supra note 8, at art. 10.1, 33 I.L.M. at 87. For a discussion of the ramifications of GATT/TRIPs for protection of computer program innovations more generally, see J.H. Reichman, The TRIPS Component of the GATT's Uruguay Round: Competitive Prospects for Intellectual Property Owners in an Integrated World Market, 4 Fordham Intell. Prop., Media & Ent. L.J. 171 (1993) [hereinafter Reichman, GATT/TRIPs].

147. Utilitarian innovations have generally been protected, if at all, by patent and trade secrecy law. See supra note 118. For the general rule against copyright protection for machines and other utilitarian works, see, e.g., 17 U.S.C. § 113(b) (1988); *Baker*, 101 U.S. at 105 (accounting system). The Supreme Court in *Baker* indicated that the principle of nonprotection for functional designs was as applicable when they were embodied in a verbal description as when they were embodied in a drawing, 101 U.S. at 103, a point that "cannot be overemphasized." Reichman, Applied Know-How, supra note 5, at 695 n.288; see also Finding a Balance, supra note 2, at 9–10 (noting that difficult questions in software copyright case law concern whether program functionality, not just code, is protectable);

Even after Congress decided to make copyright protection available to programs, their utilitarian character has made it difficult for courts to apply traditional copyright doctrines to computer programs[148] and to make distinctions that are legally as well as technically meaningful.[149] Courts have also frequently found the traditional tests for copyright infringement to be unsatisfactory or unworkable as applied to computer programs, and have consequently undertaken to develop new (and often inconsistent) tests for judging infringement in software cases.[150] However, copyright has no ready answers to many questions posed in software cases because it has never before regulated competition in technological

Samuelson, CONTU Revisited, supra note 5, at 727–53 (discussing reasons that utilitarian works have been excluded from copyright). The only machine components, before computer programs, that Congress had ever protected by copyright were sound recordings. Although sound recordings are components of machines, they are protected by copyright law because they generally embody musical works. See 17 U.S.C. § 102(a) (1988) (identifying sound recordings as among the subject matters eligible for copyright protection). As Professor Reichman explains, the bipolar structure of world intellectual property law assigns innovations in utilitarian design to patent law and innovations in literary and artistic works to copyright law. See Reichman, Legal Hybrids, supra note 100, at 2448–51. The United States tradition that conceives of the patent and copyright domains as mutually exclusive in subject matter derives from the United States Supreme Court's decision in *Baker*, 101 U.S. at 104–05. See, e.g., Taylor Instrument Cos. v. Fawley Brost Co., 139 F.2d 98, 99–100 (7th Cir. 1943), cert. denied, 321 U.S. 785 (1944). For a recent reaffirmation of this general principle by copyright scholars, see Borland Amicus Brief, supra note 5, at 8–12; see generally Overlap Study, supra note 121 (concluding that there is little if any overlap in the subject matter of patent and copyright law). There is, however, some disagreement about mutual exclusivity in the case law and in the law review literature. See, e.g., In re Yardley, 493 F.2d 1389, 1393–95 (C.C.P.A. 1974) (regarding design patent and copyright as partially overlapping in subject matter); A. Samuel Oddi, Functionality and Free Market Theory, 17 AIPLA Q.J. 173, 178–82 (1989) (suggesting that some subject matter overlap may also exist regarding utility patent and copyright law).

148. See, e.g., Computer Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 704, 712 (2d Cir. 1992) (stating that "[t]he essentially utilitarian nature of a computer program . . . complicates the task of distilling its idea from its expression"); id. at 712 (regarding much software copyright case law as "the courts' attempt to fit the proverbial square peg in a round hole"). See also Finding a Balance, supra note 2, at 22 (stating that programs' functional aspects pose difficult questions for application of copyright).

149. See, e.g., Computer Assocs. Int'l, Inc. v. Altai, Inc., 775 F. Supp. 544, 558–60 (E.D.N.Y. 1991) (discussing, with reference to a report prepared by Professor Davis, technical deficiencies with the term "structure, sequence, and organization" as applied to computer programs), aff'd in part, vacated in part, 982 F.2d 693 (2d Cir. 1992).

150. See, e.g., Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222 (3d Cir. 1986), cert. denied, 479 U.S. 1031 (1987); Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37 (D. Mass. 1990); Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203 (D. Mass. 1992) (modifying the *Paperback* test); *Altai*, 982 F.2d at 693; Gates Rubber Co. v. Bando Chem. Indus., Ltd., 9 F.3d 823 (10th Cir. 1993) (expanding *Altai* test). For further discussion of these tests, see infra notes 193–209 and accompanying text. Although Professor Miller has argued that these cases are evolving toward a consistent standard, see Miller, supra note 1, at 995–1013, the 24 copyright professors who signed an amicus brief in *Lotus v. Borland* assert that the *Whelan* and *Borland* tests cannot be reconciled with the *Altai-Gates Rubber* tests. See Borland Amicus Brief, supra note 5, at 33.

fields.[151] The very vocabulary and metaphorical structure of copyright law makes it difficult to talk about programs in a meaningful way.[152]

Copyright law is mismatched to software, in part, because it does not focus on the principal source of value in a program (its useful behavior). As explained above, program text and behavior are largely independent,[153] so that protecting program texts does not prevent second comers from copying valuable program behavior. The ability to copy valuable behavior legally would sharply reduce incentives for innovation, and thus thwart the policy behind legal protection.[154] As we explain below, the right way out of this bind is not to conceive of behavior as a nonliteral element of the program's text, as some courts have done,[155] but instead to regard it as the entity that an appropriate legal regime should protect.

Once one recognizes that computer programs are machines whose medium of construction is text,[156] it becomes obvious why copyright is an inappropriate vehicle for protecting most program behavior. Copyright law does not protect the behavior of physical machines (nor their internal construction), no matter how much originality they may embody.[157] Historically, innovations in the design of machine behavior have been left to the rigors of patent law.

Congress did not intend to protect program behavior when providing copyright protection to programs. The statutory definition of com-

151. See, e.g., Sega Enters. Ltd. v. Accolade, Inc., 977 F.2d 1510, 1513–14 (9th Cir. 1993) (whether decompiling program code to get access to interface information was an issue of first impression); *Paperback*, 740 F. Supp. at 46 (whether user interface of spreadsheet program was copyrightable treated as issue of first impression). The cases that have not been troublesome as a matter of copyright law have been those involving exact duplications of code or pictorial aspects of videogames. See, e.g., Apple Computer, Inc. v. Franklin Computer Corp., 714 F.2d 1240, 1245 (3d Cir. 1983) (exact duplication of code), cert. dismissed, 464 U.S. 1033 (1984); Stern Elecs., Inc. v. Kaufman, 669 F.2d 852 (2d Cir. 1982) (infringement of audiovisual aspects of videogame).

152. For discussion of this issue, see OTA Report, supra note 6, at 78–83; Davis, Nature of Software, supra note 5, at 314; Pamela Samuelson, Reflections on the State of American Software Copyright Law and the Perils of Teaching It, 13 Colum.-VLA J.L. & Arts 61, 73 (1988). Cases involving fabric designs, dramatic plays, or even accounting books do not provide much insight about the industrial property problems presented in computer software cases, such as whether internal interfaces of programs should be protected by copyright law. See *Altai*, 982 F.2d at 704–05 (regarding Baker v. Selden, 101 U.S. 99 (1879), as the doctrinal starting point for judging copyright infringement in cases involving utilitarian writings, but finding "scant guidance" in that case about whether a compilation of internal interface information in a computer program was protectable expression or idea; ultimately concluding it was "idea" because external factors constrained Altai's design choices).

153. See supra notes 22–24 and accompanying text.

154. See supra notes 101–108 and accompanying text.

155. See, e.g., Lotus Dev. Corp. v. Borland Int'l, Inc., 831 F. Supp. 223, 231–32 (D. Mass. 1993) (extending copyright protection to nonliteral aspects of command hierarchy).

156. See supra notes 30–42 and accompanying text.

157. See supra note 147 and accompanying text.

puter program clearly protects program texts.[158] But there is nothing in the statute nor in the legislative history to indicate that Congress intended for copyright to protect the *results* (that is, behavior) brought about by the execution of program instructions.[159]

If anything, the copyright statute would seem to preclude copyright protection for useful program behavior. Section 102(b) states, in pertinent part: "In no case does copyright protection for an original work of authorship extend to any . . . procedure, process, system, [or] method of operation . . . regardless of the form in which it is described, explained, illustrated, or embodied in such work."[160] According to Professor Miller, Chair of the National Commission on New Technological Uses of Copyrighted Works (CONTU) subcommittee responsible for recommending copyright protection for computer software,[161] the CONTU Commissioners believed "that copyright should not block anyone from developing their own software, even if that software were to perform the same function as pre-existing software *and were to accomplish it in the same way*."[162] Some recent cases have rejected copyright claims for the behavior of programs on the ground that behavior is among the processes made unprotectable by section 102(b).[163]

That program behavior, in general, is unprotectable by copyright law on account of its functionality does not mean that behavior can never be protected by copyright law. Sometimes program behavior is "expressive" in a traditional copyright sense; when it is, copyright applies. Recall that program text is a medium in which it is possible to construct both useful and expressive behaviors.[164] Just as it is possible to construct both useful devices (such as chairs or screwdrivers) and expressive devices (such as sculptures or music boxes) out of media such as steel and plastic, it is

158. See 17 U.S.C. § 101 (1988) (defining "'computer program' [as] a set of statements or instructions to be used directly or indirectly in a computer in order to bring about a certain result").

159. See H.R. Rep. No. 1307, 96th Cong., 2d Sess., pt.1, at 23 (1980), reprinted in 1980 U.S.C.C.A.N. 6460, 6482 (citing CONTU Report as source for new legislation with no additional discussion to indicate addition of copyright protection for program results); Borland Amicus Brief, supra note 5, at 5–12 (functional processes brought about by the execution of program instructions unprotectable by copyright); CONTU Report, supra note 17, 37–46 (discussing program text); see also Paul Goldstein, Infringement of Copyright in Computer Programs, 47 U. Pitt. L. Rev. 1119, 1124–26 (1986) (regarding manner in which programs perform their functions as unprotectable by copyright).

160. 17 U.S.C. § 102(b) (1988).

161. See CONTU Report, supra note 17, at 10.

162. Declaration of Arthur R. Miller at 4, Evergreen Consulting, Inc. v. NCR Comten, Inc., No. CV 82-5946 (C.D. Cal. Jan. 3, 1985) (emphasis added) [hereinafter Miller Declaration]; see also CONTU Report, supra note 17, at 39–41.

163. See, e.g., Gates Rubber Co. v. Bando Chem. Indus., 9 F.3d 823, 836 (10th Cir. 1993); Computer Assocs. Int'l v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991), aff'd in part, vacated in part on other grounds, 982 F.2d 693 (2d Cir. 1992). But see Lotus Dev. Corp. v. Borland Int'l, Inc., 831 F. Supp. 223 (D. Mass. 1993) (regarding program behavior as "nonliteral" aspect of program text protectable by copyright if alternatives possible).

164. See supra text accompanying note 43.

possible to construct both useful behaviors and expressive behaviors out of software.

United States law has devised imperfect, yet nonetheless workable, ways to distinguish between a copyrightable sculpture and an uncopyrightable tool when dealing with artifacts made of steel or plastic.[165] It can make similar distinctions between the expressive and functional behavior of computer programs. That it may occasionally be difficult to distinguish between a useful artifact and an expressive artifact (e.g., a kinetic sculpture) does not make easy cases impossible to recognize. Nor does it undermine the usefulness of a general distinguishing principle of this sort.

When the execution of program instructions results in the display of pictures or text that qualify as an original work of authorship separate from the program that generated it, copyright protection for the picture or text is appropriate.[166] When the execution of program instructions results in a series of pictorial images combined with text and sounds, program behavior can be part of a copyrightable audiovisual work.[167]

When program behavior has both functional and expressive components, it should generally be easy to distinguish between them for copyright purposes. The primary function of screen-saver programs, for example, is to produce behavior that protects computer screens from burnout. Screen-saver behavior may also include moving pictures of flying toasters. If a second comer copies the flying toaster graphics, copyright infringement is likely.[168] Courts should not, however, find copyright infringement if a second comer develops a program that protects

165. See, e.g., Brandir Int'l, Inc. v. Cascade Pac. Lumber Co., 834 F.2d 1142, 1145 (2d Cir. 1987) (holding that derivative design of copyrighted sculpture was uncopyrightable because functional and artistic design elements were inseparable in resulting bicycle rack); Esquire, Inc. v. Ringer, 591 F.2d 796 (D.C. Cir. 1978), cert. denied, 440 U.S. 908 (1979) (holding overall shape or configuration of utilitarian article noncopyrightable, no matter how unique or aesthetically pleasing). For a thorough discussion of the difficulties of distinguishing between applied art and industrial design, see Reichman, Design Protection 1948–1976, supra note 54 (discussing regulation of ornamental designs in useful articles); Reichman, Design Protection After 1976, supra note 54 (discussing response of courts and administrators to dilemmas in design protection).

166. See, e.g., Computer Assocs. Int'l v. Altai, Inc., 982 F.2d 693, 703 (2d Cir. 1992) (indicating that results generated by execution of programs' instructions must be independently copyrightable for similarities to form basis of infringement claim).

167. See, e.g., Stern Elec., Inc. v. Kaufman, 669 F.2d 852 (2d Cir. 1982) (audiovisual copyright in videogame's repetitive sequence). Some aspects of a videogame's behavior are not protectable by copyright law. See, e.g, Data East USA, Inc. v. Epyx, Inc., 862 F.2d 204, 209 (9th Cir. 1988) (holding elements of karate videogame "subject to the constraints inherent in the sport" unprotectable by copyright).

168. Berkeley Software brought suit against the developer of a competing screen saver program that parodied the Berkeley "After Dark" flying toaster graphics. Ironically, Jefferson Airplane recently sued Berkeley for copying flying toasters from one of Airplane's record album covers. See Jim Doyle, Jefferson Airplane Files Copyright Suit, S.F. Chron., June 15, 1994, at A14.

computer screens from burnout in a manner that is functionally indistinguishable from the first program.[169]

Although neither CONTU nor Congress specifically stated that copyright should protect expressive program behavior, such protection is consistent with the copyright statute. Section 102(a) states that "[c]opyright protection subsists . . . in original works of authorship fixed in any tangible medium of expression, now known or later developed."[170] Original works, such as text and pictures, that have been fixed in the medium of program code should qualify for copyright protection just as if they had first been fixed in print or on canvas.[171]

If the industrial designs of programs cannot generally be protected by patents because they do not meet the patent invention standard, but need some legal protection, one might ask whether copyright law could supply this protection. The originality standard of copyright law would not pose a problem for most industrial design elements of programs because program development requires creativity.[172]

However, by enacting section 102(b),[173] Congress intended to exclude industrial design elements of programs—processes, procedures, systems, and methods of operation—from the scope of copyright protection, as is evident from the legislative history of this provision. The House and Senate Reports explain:

> Some concern has been expressed lest copyright in computer programs should extend protection to the methodology or processes adopted by the programmer rather than merely to the "writing" expressing his ideas. Section 102(b) is intended, among other things, to make clear that the expression adopted by the programmer is the copyrightable element in a computer program, and that the actual process or methods embodied in the program are not within the scope of copyright law.[174]

At virtually all levels of abstraction above the literal text, a program consists of procedures, processes, systems, methods of operation, or their components. The exclusion of some industrial design elements of pro-

169. Some courts have nonetheless found copyright infringement in these circumstances. See, e.g., Lotus Dev. Corp. v. Borland Int'l, Inc., 831 F. Supp. 223 (D. Mass. 1993). For a more complete discussion, see infra notes 172–209 and accompanying text.

170. 17 U.S.C. § 102(a) (1988).

171. In his concurrence to the CONTU Report, Professor Nimmer suggested that, in time, Congress might decide to use copyright law only to protect those programs that produced copyrightable works. See CONTU Report, supra note 17, at 27. He also foresaw the danger that protecting programs by copyright law might have the effect of converting copyright law to a general misappropriation law, intruding on the patent domain, and posing serious constitutional and policy questions. See id. at 26.

172. For copyright's originality requirement, see Feist Publications, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340 (1991) (holding telephone book to lack modicum of creativity required to meet "originality" requirement).

173. 17 U.S.C. § 102(b) (1988).

174. H.R. Rep. No. 1476, 94th Cong., 2d Sess. 57 (1976), reprinted in 1976 U.S.C.C.A.N. 5659, 5670; S. Rep. No. 473, 94th Cong., 1st Sess. 54 (1976).

grams from the scope of copyright is consistent with longstanding principles of both copyright and patent law, and competition policy objectives that underlie these laws.[175]

Proponents of a broader scope of copyright protection for computer programs tried to persuade both CONTU and Congress to support an amendment to section 102(b) that would have allowed copyright protection for "the logical choices made by those personnel working on the design phase of program development."[176] The amended section 102(b) would have provided: "However, copyright protection may exist in a collection of ideas or abstractions, arbitrarily selected from a plurality of alternative ideas or abstractions or in a discretionary pattern of events or processes."[177] Professor Miller reports that CONTU rejected this proposal "because it would have been inconsistent with the traditional nature of copyright protection for particular forms of expression and not for underlying processes."[178] CONTU also "declined to adopt a recommendation by proponents of copyright for the logic of computer operations that would have amended the Copyright Act to provide that '[a] computer program may be a derivative work of a flow chart and either may be a derivative work of a literary work.' "[179]

The reason CONTU rejected these proposals was simple:

175. See supra notes 18–21 for the nature of programs. For discussions of policy, see Karjala, supra note 1, at 36–41; Arthur J. Levine, Comment on *Bonito Boats* Follow-up: The Supreme Court's Likely Rejection of Nonliteral Software Copyright Protection, Computer Law., July 1989, at 29; Menell, supra note 1, at 1081; D.C. Toedt, *Bonito Boats* Follow-up: Free Competition Public Interests vs. the Substantial Similarity Test—Does The Legislative History Actually Support Nonliteral Software Copyright Protection, Computer Law., July 1989, at 14.

The CONTU Report, on which Congress relied in enacting the computer-program-related amendments to the copyright statute, seems to have contemplated that copyright would protect program code, because all of the examples of infringing acts described in the Commission's report were examples of exact copying. See CONTU Report, supra note 17, at 21–23. The Commission's stated economic rationale for using copyright to protect programs also focused on exact copying. See id. at 11. CONTU referenced 17 U.S.C. § 102(b) as among the limiting provisions and doctrines of copyright law that would apply to programs. See id. at 18–20.

176. Second Declaration of Arthur R. Miller at 4, Evergreen Consulting, Inc. v. NCR Comten, Inc., No. CV 82-5946 (C.D. Cal. June 24, 1985) [hereinafter Second Miller Declaration].

177. Id. at 4 n.2.

178. Id. at 4. Miller observed that the tradition of not protecting such processes went back to Baker v. Selden, 101 U.S. 99 (1879), and served to differentiate computer programs from fanciful works. See Second Miller Declaration, supra note 176, at 7.

179. See Second Miller Declaration, supra note 176, at 4. Had such a provision been enacted, Miller noted, it "could have effectively granted monopolies over logical processes through the registration of copyrights on flow charts, irrespective of whether such processes would meet patentability requirements." This was not consistent, Miller insisted, with what Congress intended by enactment of § 102(b), nor with the public policies underlying patent law, which protects only nonobvious advances in the technological arts, and then only if patent requirements have been satisfied. See id. at 4–5.

> To argue that copyright protection should be afforded to computer program processes that are not novel enough to satisfy the statutory requirements for patent protection because they are the result of creative effort and the investment of substantial resources, is to voice what may be a legitimate concern in the wrong legal environment. The Copyright law is not the forum for determining how to deal with such a problem, assuming one exists.[180]

If there is a gap in the legal protection regime for computer software, Miller seemed to say, it should emanate from Congress, not by wrenching copyright law beyond its statutory bounds.[181]

This result does not change if one characterizes the abstract design of programs as an industrial compilation of applied know-how. Of course, copyright law has a long history of protecting compilations of information.[182] It has, however, accorded either no or relatively thin protection to compilations of industrial or technological elements.[183] Many industrial compilations (such as a combination of machine components) have been excluded from copyright protection because of their intrinsic utility.[184] Others, such as those created when someone selects or arranges information in accordance with a method, system, or other functionally determined design, are also unprotectable by copyright law.[185] Even when copyright protection has been available for a work consisting largely of industrial information (such as a manual for operating a power plant), it has not protected the know-how, only the author's manner of describing the know-how.[186] The commercially valuable aspects of industrial compilations of applied know-how have generally been protected by trade secret, not copyright.[187]

---

180. Id. at 12.

181. Miller has recently changed his views about copyright protection for programs. He now argues that copyright protection should extend to creative aspects of programs in order to create the proper incentives for software development. See Miller, supra note 1, at 1006–10. This does not, however, alter the historical fact that CONTU and Congress rejected proposals to make all discretionary program design choices protectable by copyright law. Interestingly, Miller's recent article on software protection does not discuss section 102(b) of the copyright statute or the long line of cases holding that the functional designs embodied in copyrighted works are unprotectable by copyright law. For a partial listing of such cases, see Samuelson, Critique of *Paperback*, supra note 5, at 324 n.61.

182. The first United States copyright statute permitted copyright protection for books, maps, and charts. See Ginsburg, supra note 74, at 1873–88 (discussing long history of copyright protection for compilations of information).

183. See Reichman, Applied Know-How, supra note 5, at 691–92.

184. See supra note 147.

185. See Baker v. Selden, 101 U.S. 99, 103 (1879) (accounting forms); Kregos v. Associated Press, 937 F.2d 700, 703–10 (2d Cir. 1991) (baseball pitching statistics form). Patents may sometimes be available for inventive methods or systems of this sort. See Chisum, Algorithms, supra note 1, at 1021–22 (giving examples of patents on functional information innovations).

186. See *Baker*, 101 U.S. at 104.

187. See supra note 118.

If program compilations are too technological to be protected by copyright, one might still argue for copyright protection based on the line of cases protecting "sweat of the brow." Until recently, American copyright case law extended copyright protection to factual compilations lacking creativity in selection or arrangement of elements, on the theory that without protection by copyright, there would be too little incentive to invest in the production and dissemination of these useful works.[188] If these "sweat of the brow" cases were still viable precedents, they would provide a basis for copyright protection of the compilations of applied know-how in programs. However, the Supreme Court, in *Feist Publications, Inc. v. Rural Telephone Service Co.*,[189] rejected the "sweat of the brow" rationale for extending copyright protection to compilations that lacked expressive originality.

2.3 Use of Traditional Legal Regimes to Protect Software Innovations Will Lead to Cycles of Under- and Overprotection

In a report to the Congress on intellectual property protection for computer software, the Office of Technology Assessment once stated that if copyright did not protect more than the literal code of computer programs, it would protect too little, and if it protected more than the literal code, it would protect too much.[190] We agree and make a similar point about patents: Most of the commercially significant innovations in software will be underprotected if patent law adheres to its traditional bounds; yet if this law is stretched to protect commercially valuable program innovations, it will overprotect them.[191]

Professor Reichman has shown that use of classical intellectual property regimes to protect other dual-nature works, such as industrial designs of manufactured products, has produced recurrent cycles of under- and overprotection over a two-hundred-year period.[192] These cycles generally start when legal authorities resist efforts to extend the bounds of existing law. This brings about underprotection. Faced with the market-destruc-

188. See Ginsburg, supra note 74, at 1909–13.

189. 499 U.S. 340, 353–54 (1991).

190. OTA Report, supra note 6, at 78; see also Reichman, Applied Know-How, supra note 5.

191. Underlying the effort to make copyright law into a trade secrecy protection law, see infra notes 324–330 and accompanying text, have been concerns about under- and overprotection for valuable aspects of software. If the effort to use copyright law to prevent intermediate copying of programs in order to get access to internal design elements had succeeded, software would have been accorded a 75 year exception to the general trade secrecy rule that reverse analysis of mass-marketed products is lawful. See 17 U.S.C. § 302 (1988) (setting forth duration of copyright for works published by firms). This duration would have been out of proportion to any harmful market effects that could have arisen from such reverse analysis. Yet, without copyright to protect internal design elements as trade secrets, these innovations are likely to be underprotected by existing regimes. See Reichman, Overlapping Rights, supra note 5.

192. See Reichman, Design Protection 1948–1976, supra note 54, at 1143.

tive effects of underprotection, legal authorities have sometimes been persuaded to extend the boundaries of existing regimes to protect the dual-natured innovations before them. Although such boundary extension may solve one developer's immediate underprotection problem, it often hurts the market as a whole by overprotecting incremental technical innovations embodied in publicly distributed products.

The remainder of this subsection will demonstrate that cycles of under- and overprotection of software innovations by existing legal regimes are underway in American law. To avoid these cycles, a new legal regime is needed that protects industrial design elements in programs from trivial acquisitions of functional equivalence.

2.3.1 Use of Copyright to Protect Industrial Designs Leads To Under- and Overprotection

Within the traditional framework of copyright law, judges have two equally unpalatable options when dealing with a defendant who has appropriated commercially valuable industrial design elements from another firm's program: either not protecting those design elements at all, or protecting them for seventy-five years.[193] This is a choice between underprotection and overprotection. Cycles of under- and overprotection occur as courts oscillate between extending broad protection out of concern that software will otherwise be underprotected by the law and permitting valuable designs to be freely copied in accordance with well-established limiting doctrines of copyright law.

Whelan Associates v. Jaslow Dental Laboratory, Inc. is the clearest example of a decision in which the prospect of underprotection led a court to construe the scope of copyright protection for programs very expansively.[194] Relying on evidence indicating that the design of software was

193. See 17 U.S.C. § 302(c) (1988) (providing that works authored by firms receive copyright protection for 75 years from the date of publication or 100 years from creation, whichever is less). Occasionally, a software developer will be lucky enough to enjoy both the peak and the valley of these cycles. Compare Synercom Technology, Inc. v. University Computing Co., 462 F. Supp. 1003 (N.D. Tex. 1978) (holding that input formats are unprotectable by copyright) (Engineering Dynamics, the principal defendant) with Engineering Dynamics, Inc. v. Structural Software, Inc., 26 F.3d 1335 (5th Cir. 1994) (ruling that case law had evolved since *Synercom* and that input formats may now be protectable by copyright).

194. 797 F.2d 1222 (3d Cir. 1986), cert. denied, 479 U.S. 1031 (1987). Whelan, like other software copyright plaintiffs, argued that computer programs are "literary works" within the meaning of the copyright statute, and if copyright protects the "structure, sequence, and organization" (SSO) of other literary works as "nonliteral elements" of their text, then surely the SSO of programs can also be protected by copyright law. See id. at 1234. Even the Second Circuit's *Altai* decision which is otherwise critical of *Whelan*, see infra note 201, found this syllogism persuasive. See Computer Assocs. Int'l v. Altai, Inc., 982 F.2d 693, 702 (2d Cir. 1992). Of course, an article reporting a mathematical or scientific breakthrough is also a "literary work" within the meaning of 17 U.S.C. § 101. Yet, the logic and structure of the mathematical formula or scientific analysis described in the article would not be within the scope of copyright under 17 U.S.C. § 102(b). Nor, if the

the most creative, costly, and valuable of its aspects and that design needed protection if software developers were to have sufficient incentives to invest in innovative designs,[195] this court decided that everything about a program, except its general purpose or function, was protectable "expression."[196] If, however, there were only one or very few ways to perform a function, the court would consider "idea" and "expression" to be "merged" and no copyright protection would be available for the merged expression.[197]

---

article was about a new industrial process, would a copyright in the article protect the structure, sequence, and organization of the process, under the principles of Baker v. Selden, 101 U.S. 99 (1879), and its progeny.

Some commentators have likened computer programs to "poetry" hoping that this would persuade courts to grant the broad scope of copyright protection that artistic and fanciful works have generally enjoyed, rather than the thin scope of protection typically accorded to functional writings. See, e.g., Clapes et al., supra note 1, at 1497, 1504 (quoting computer scientist Frederick Brooks who likened programmers to poets); Miller, supra note 1, at 984, 995 (comparing communicative precision required of a programmer to that of a poet working within constraints of iambic pentameter).

Frederick Brooks, however, says he gave up the metaphor of "writing" in relation to programs in 1958 in favor of the metaphor of "building" programs. See Brooks, supra note 36, at 18. Brooks also endorses the use of software engineering techniques to improve the industrial design of programs. See id. As we have explained above, supra note 73 and accompanying text, it is accurate to characterize programmers as engineers. It does not denigrate the creativity required to do such tasks to refer to programs as products of incrementally innovative engineering. It does, however, limit the scope of copyright protection.

195. See *Whelan*, 797 F.2d at 1231, 1237.

196. See id. at 1236.

197. See id. at 1236. The "merger" doctrine is a well-established principle of United States copyright law. See Paul Goldstein, Copyright Principles, Law & Practice § 2.3.2 (1989). Professor Menell has proposed using the merger doctrine to avoid the economically harmful consequences of overbroad decisions, such as *Whelan*. He suggests that courts apply the doctrine whenever network externalities make some aspect of a program the de facto industry standard, such that other competitors need to reproduce it to compete effectively, or when a component of a program is the most efficient means of performing its function. See Menell, supra note 1, at 1050, 1084–88.

Some judges have approximated this result by regarding user expectations as "external factors" constraining the design choices of subsequent developers. See, e.g., Plains Cotton Coop. Ass'n v. Goodpasture Computer Serv., Inc., 807 F.2d 1256, 1262 (5th Cir.), cert. denied, 484 U.S. 821 (1987). Other judges, most notably Judge Keeton in the *Lotus* cases, have rejected these arguments. Judge Keeton believes that punishing successful software developers by forcing them to contribute functionally optimal or de facto standard designs to the public domain would subvert the basic purpose of copyright. See Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 79 (D. Mass. 1990).

While we think Menell's proposal is better, as a matter of copyright law, than the overbroad *Whelan-Paperback* approach, we agree with Judge Keeton that functionally optimal software designs and de facto standards should enjoy some degree of protection under an appropriate legal regime for software innovations. See infra notes 368–371 and accompanying text.

In attempting to save the software industry from underprotection, the *Whelan* court inadvertently thrust it into overprotection,[198] and contravened the explicit intent of Congress.[199] The *Whelan* approach would treat as "expression" every discretionary choice made during the design phase of program development, as though Congress had adopted the proposed amendments to section 102(b) that it and CONTU had, in fact, rejected.[200]

The case that turned the tide against the expansive *Whelan* approach was the 1992 Second Circuit Court of Appeals decision in *Computer Associates International, Inc. v. Altai, Inc.*[201] The court in *Altai* recognized that programs were utilitarian works, and under traditional principles of copyright case law such as those reflected in *Baker v. Selden*[202] and the statu-

198. The *Whelan* decision initially attracted some support from other courts. See, e.g., *Paperback*, 740 F. Supp. at 52; Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1133 (N.D. Cal. 1986). But see *Plains Cotton*, 807 F.2d at 1262. This support eroded as courts began to realize, in part because of a flood of critical commentary about *Whelan* and its test for software copyright infringement, that the *Whelan* approach would overprotect innovation in software. For a partial listing of the law review criticism of *Whelan*, see, e.g., Borland Amicus Brief, supra note 5, at 28 n.55. For recent decisions critical of *Whelan*, see infra note 201 and accompanying text.

199. See supra notes 173–181 and accompanying text regarding 17 U.S.C. § 102(b) and the congressional concern about overprotection of program innovations by copyright law that motivated its enactment.

The court in *Whelan* did not even try to distinguish between the "structure, sequence, and organization" of programs that it thought copyright law could protect, and the processes, procedures, systems, and methods of operation that 17 U.S.C. § 102(b) rendered unprotectable by copyright.

Not only did *Whelan* ignore the basic copyright policy, it also undermined the economic policies underlying patent law. Protecting unpatentable processes and the like through copyright law would contravene the economic policies underlying patent law, for it would give a very long period of exclusive rights to innovations that have not undergone the rigorous examination process required by the patent system nor met the patent standards. See, e.g., Dennis S. Karjala, Recent United States and International Developments in Software Protection (pt. 1), 16 Eur. Intell. Prop. Rev. 13, 14 (1994); Menell, supra note 1, at 1049–50.

200. See supra notes 176–181 and accompanying text.

201. 982 F.2d 693 (2d Cir. 1992). The court in *Altai* criticized *Whelan* for taking an overbroad view about the scope of protection available to programs from copyright law, for having an outmoded understanding of computer science, and for relying too much on metaphysical distinctions rather than practical realities. See id. at 705–06. The court also criticized arguments that programs should be given a broad scope of protection by copyright lest there be insufficient incentives to invest in software development. It regarded these arguments as inconsistent with the Supreme Court's decision in Feist Publications, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340 (1991). See *Altai*, 982 F.2d at 711–12. Among the cases endorsing *Altai* are Gates Rubber Co. v. Bando Chem. Indus., Ltd., 9 F.3d 823, 841 (10th Cir. 1993); Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1524–25 (9th Cir. 1992); and Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832, 839 (Fed. Cir. 1992). Other appellate decisions that are more consistent with *Altai* than with *Whelan* are: Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir.), cert. denied, 113 S. Ct. 198 (1992); and *Plains Cotton*, 807 F.2d at 1256.

202. 101 U.S. 99 (1879). *Altai* spoke of *Baker v. Selden* as "the doctrinal starting point" in analysis of cases involving utilitarian works, such as computer programs. *Altai*, 982 F.2d

tory exclusion of processes and systems under section 102(b),[203] programs—like other utilitarian works—should enjoy, at most, a narrow scope of copyright protection. The court also identified three classes of elements that should be "filtered out" as unprotectable material before making a determination of infringement: elements that are dictated by efficiency; elements that are constrained by external factors, such as the hardware or software with which the program needed to be compatible; and elements common in the industry or otherwise in the public domain.[204] The court reasoned that these elements were beyond the scope of protection that copyright law provides to qualifying works of authorship, so that others should be free to use them in subsequent works.[205]

As the court seemed to acknowledge in *Altai*, filtering out all of the unprotected elements may leave little besides the program's text as protectable expression.[206] It responded to incentive-based arguments for stretching copyright law to protect all of the valuable nonliteral elements of programs by saying that while the arguments had some appeal from a pure policy standpoint, ultimately they would "have a corrosive effect on

at 704. See also *Sega*, 977 F.2d at 1524 (also discussing the importance of *Baker v. Selden* in cases involving works with a high degree of functional content).

203. See *Altai*, 982 F.2d at 704–05. For a discussion of the implications of *Baker v. Selden* for understanding the exclusions from the scope of copyright set forth in 17 U.S.C. § 102(b), see Reichman, Applied Know-How, supra note 5, at 693–95 n.288; Samuelson, Critique of *Paperback*, supra note 5, at 325–36; see also Borland Amicus Brief, supra note 5, at 12–19 (asserting that Congress enacted § 102(b) to embody principles of *Baker*, including notion that utilitarian systems and processes are uncopyrightable).

204. See *Altai*, 982 F.2d at 706–12. See supra note 58 concerning efficiency as a constraint on design choices. Although we agree that these elements of programs should be filtered out in the final stage of an infringement determination, we note that the *Altai* decision seems to begin with much the same assumption as *Whelan*: that "nonliteral" elements of programs should be treated as "expression," absent some limiting principle or doctrine of copyright law that filters them out of infringement consideration. Because abstract elements of programs are so frequently components of the processes and procedures implemented by the programs, the more appropriate assumption would be to regard them as unprotectable processes unless shown to be expressive. Interestingly, *Altai* did not expressly direct filtration of processes, procedures, systems and methods of operation embodied in programs, as 17 U.S.C. § 102(b) seems to require. In some decisions following *Altai*, courts have also begun filtering out the processes, procedures, systems, and methods of operation that section 102(b) excludes from copyright protection. See, e.g., *Gates Rubber*, 9 F.3d at 836; Apple Computer, Inc. v. Microsoft Corp., 799 F. Supp. 1006, 1021 (N.D. Cal. 1992), aff'd, 35 F.3d 1435 (9th Cir. 1994). See supra notes 173–181 and accompanying text for a discussion of section 102(b).

205. See *Altai*, 982 F.2d at 707–10; see also Feist Publications, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340, 361–64 (10th Cir. 1991) (affirming right of public to copy unprotected matter from copyrighted works, including incorporation in subsequent works).

206. See *Altai*, 982 F.2d at 712 ("[I]t may well be that the Copyright Act serves as a relatively weak barrier against public access to the theoretical interstices behind a program's source and object codes.") The *Altai* decision brought to the surface a new round of concerns about underprotection of program innovations by copyright law. See, e.g., John Markoff, Ruling Restricts Copyright Protection, N.Y. Times, June 24, 1992, at D1; Jonathan M. Moses, Court Eases Copyright Rules For Software, Wall St. J., June 24, 1992, at B1.

certain fundamental tenets of copyright doctrine."[207] Professor Karjala has pointed out that some of the decisions purporting to follow *Altai* have tended to do less filtration than *Altai* would seem to require.[208] We suggest that they have done this out of concern that program innovations would be legally underprotected if the courts took filtration seriously.[209]

2.3.2 Cycles of Under- and Overprotection Have Also Been Evident in Patent Law

Concerns about overprotection contributed to the initial policy denying patents for software innovations.[210] A 1966 Presidential Commission recommended against patent protection for program innovations, in part, because programs were already protected by copyright law.[211] It noted that progress in this field was occurring without patents,[212] and was also concerned that the Patent Office would be unable to make good judgments about software-related applications because it did not have a repository on the state of this art, appropriately trained examiners, or an adequate classification system for software innovations, as well as because it would be flooded with applications it did not have the resources to handle.[213]

The Supreme Court also expressed concern about overprotection when it rejected patent claims for a program algorithm to convert binary coded decimals to pure binary form.[214] The Court worried that patents on algorithms would impede the widespread use of mathematical ideas, which, like scientific principles and laws of nature, had long been re-

207. *Altai,* 982 F.2d at 712. The court went on to say that "[i]f the test we have outlined results in narrowing the scope of protection, as we expect it will, that result flows from applying, in accordance with Congressional intent, longstanding principles of copyright law to computer programs." Id. The court expressed concern "that these fundamental principles remain undistorted." Id.

208. See Dennis S. Karjala, Recent United States and International Developments in Software Protection (pt. 2), 16 Eur. Intell. Prop. Rev. 58, 60–62 (1994).

209. For a recent airing of such concerns, see, e.g., Julian Velasco, Note, The Copyrightability of Nonliteral Elements of Computer Programs, 94 Colum. L. Rev. 242, 285 (1994).

210. See supra notes 121–130 and accompanying text concerning the initial policy on patenting software innovations.

211. See President's Comm'n on the Patent System, "To Promote the Progress of... Useful Arts" in an Age of Exploding Technology 13 (1966) [hereinafter 1966 Report].

212. See id. This suggests that the Commission thought that patents might be disruptive to that progress.

213. See id. A Patent Office ill-equipped to make good judgments on software-related applications would likely issue many "bad" patents, that is, patents on ideas that, unbeknownst to the Patent Office, were already in the state of the art, were only modest advances in the art, or needed to be claimed more narrowly in view of the prior art. "Bad" patents are overprotective.

214. See Gottschalk v. Benson, 409 U.S. 63, 72–73 (1972). The Court quoted at length from the 1966 Report's recommendations against patents for software. See id. at 72.

garded as unpatentable subject matter.[215] Also, because algorithms sometimes have an extremely wide array of potential applications, including ones never envisioned by the discoverer of the algorithm, the Court worried that patents for algorithms would overprotect these innovations.[216]

As the software industry became more commercially significant, the early concerns about possible overprotection of program innovations through patents subsided and were replaced by concerns that the initially constrictive patent policy toward software innovation might, in fact, underprotect valuable aspects of programs.[217] By the mid-1980s, the Patent and Trademark Office had begun to give an increasingly broad construction to the patent subject matter provisions. This was partly as a result of pressure from some industry sectors—mainly computer companies, not those who considered themselves "software publishers"[218]—and partly as a result of several appellate court decisions that narrowly construed the Supreme Court precedents on software-related patents.[219] Today, the Patent and Trademark Office seems to regard almost all program-related innovations, except perhaps those claiming "*mathematical* algorithms and abstract mathematical formulae," as patentable subject matter.[220]

215. See id. at 67.

216. See id. at 67, 71; see also Parker v. Flook, 437 U.S. 584, 595–96 (1978) (rejecting claim for industrial application of mathematical algorithm as unpatentable subject matter; finding no congressional intent to protect such innovations by patent law).

217. See Chisum, Algorithms, supra note 1, at 1013–20.

218. Major computer companies, such as IBM, were initially opposed to patent protection for software innovations, such as algorithms. See *Gottschalk*, 409 U.S. at 63 (indicating briefs amici curiae filed in support of Patent Office's position on behalf of Burroughs Corp., Honeywell, Inc., International Business Machines Corp., and Business Equipment Manufacturers Ass'n). This position was consonant with their business interests at the time. As hardware vendors, they supplied software with their machines only as a way to make the hardware more desirable. Patents might have required them to pay royalties on innovations that, in effect, they gave away with hardware. As the market for software evolved and computer firms began to perceive how lucrative the software business could be, they became some of the most ardent supporters of software patents. See John P. Sumner & Steven W. Lundberg, Software Patents: Are They Here To Stay?, 8 Computer Law., Oct. 1991, at 8, app. at 13–15 (presenting views of computer industry lawyers favoring patents for software innovations). The computer firms have been active in recent years obtaining software-related patents. See John T. Soma & B. F. Smith, Software Trends: Who's Getting How Many of What? 1978 to 1987, 71 J. Pat. & Trademark Off. Soc'y 415 app. at 428–32 (1989) (showing that hardware firms had obtained most of the software patents).

Software developers have generally been more doubtful of the appropriateness of patents for software innovations. See, e.g., Glen D. Self, Comments on Behalf of the Research and Development Organization of EDS, in Response to Request for Comments for the Advisory Commission of Patent Law Reform, 56 Fed. Reg. 22702-02 (1991) (on file with the Columbia Law Review) and sources cited infra note 221.

219. See, e.g., In re Iwahashi, 888 F.2d 1370 (Fed. Cir. 1989). For an argument that *Iwahashi* cannot be squared with *Benson*, see Samuelson, *Benson* Revisited, supra note 5, at 1102 n.294.

220. *Iwahashi*, 888 F.2d at 1374.

While some still oppose algorithm patents on the ground that they interfere with the free use of mathematical ideas,[221] the major battleground over software-related patents today concerns the potential for overprotection of program innovations arising from deficiencies in the Patent and Trademark Office's handling of software-related applications.[222] As the 1966 Commission feared, the Patent and Trademark Of-

221. See, e.g., Report of the Committee on Algorithms and the Law, Optima (Mathematical Programming Society), June 1991, at 2, 4 [hereinafter Mathematical Report] ("[W]e believe that the patenting of algorithms would have an extremely damaging effect on our research and on our teaching, particularly at the graduate level, far outweighing any imaginable commercial benefit."). The Patent and Trademark Office recently sponsored two hearings at which software developers voiced their concerns about or opposition to the use of patents in regulating innovation and competition in the software industry. See, e.g., Gregory Aharonian <srctran@world.std.com>, PTO Software Patent Hearings in DC, Internet Pat. News Serv., Feb. 12, 1994, URL: http://sunsite.unc.edu/patents/intropat.html (archived stories); United States Patent and Trademark Office Public Hearing on Use of the Patent System to Protect Software-Related Inventions: Before Bruce A. Lehman, Assistant Secretary of Commerce and Comm'r of Patents and Trademarks, San Jose, Cal., Jan. 27, 1994 (statement of Jim Warren, Director, Autodesk, Inc.) [hereinafter Patent Hearings]; see also Patent Hearings, supra, Jan. 26, 1994 (statement of Tom Lopez, President, Interactive Multimedia Ass'n). The Interactive Media Association opposes patents pertaining to multimedia products because patents would impede free expression and substantially increase development costs, especially the cost of insurance against unwitting infringement of patents. It is interesting to observe that although there is very little critical commentary about patents for software innovations in the law review literature, there is a good deal of it in non-legal periodicals. See, e.g., Peter Coffee, Soft Talk: Rulings Show Absurd State of Software Law, PC Wk., Apr. 18, 1994, at 58 (arguing against patents for software); Jonathan Erickson, Baby Don't You Drive My Car, Dr. Dobb's J., Jan. 1991, at 96S; Michael J. Miller, Software Patents Must Go, PC Mag., Mar. 15, 1994, at 79 (discussing litigation issues); Dana J. Parker, Patently Ridiculous, CD-ROM Prof., May 1994, at 161 (discussing "good" and "bad" standards); Evan I. Schwartz & Michele Galen, The Coming Showdown over Software Patents, Bus. Wk., May 13, 1991, at 104; Software Patents: Law of the Jungle, Economist, Aug. 18, 1990, at 59; Softwars, Economist, Apr. 23, 1994, at 17; Patent Law: The Strange Case of Dr. Billings, Economist, Oct. 16, 1993, at 82.

222. As the 1966 Report feared, the Patent and Trademark Office began issuing patents in this field without an adequate repository of the art in the field (a deficit which is far more serious today than it was in 1966), without adequately trained examiners, and without an adequate classification system. After almost a decade of granting patents for software-related innovations, it has still not resolved these problems. The administrative difficulties are so severe that a recent report of the Office of Technology Assessment on the state of software intellectual property law identified reform of the Patent and Trademark Office's practices as "extremely important." Finding a Balance, supra note 2, at 24. See generally id. at 6–35 (discussing administrative problems with patenting software innovations). The current Commissioner of Patents and Trademarks, Bruce Lehman, has spoken of the Patent and Trademark Office administration of software-related patent applications as an "indefensible mess." Bob Metcalf, From the Ether: Washington Grinds Down New Patent Chief, InfoWorld, May 9, 1994, at 62. The Patent and Trademark Office has recently undertaken sua sponte to reexamine some software-related patents that generated almost universal outrage from the industry as wrongly issued. See, e.g., Mike Landberg, Compton's Newmedia Loses Controversial Patent, San Jose Mercury News, Mar. 26, 1994, at 12D.

fice has seen a significant influx of applications, and has issued sharply increasing numbers of patents.[223]

Although the Office is improving its practices,[224] its reforms cannot solve the principal protection problem that this Article addresses. Because innovation in the software industry is typically incremental, not inventive, the success of patent reform will inevitably withdraw most patent protection from software and result in underprotection.[225]

2.4 Reasons Why Action Is Needed

Several considerations motivate our call to consider a new framework for protecting software innovations in the near term. First, as we have argued above, there is a substantial source of value in programs that is easily imitated for which existing law, properly construed, provides no remedy.[226] As an information product, software has little or no lead time deriving from mass production or distribution. As a result, innovators may have little lead time left if second comers can quickly make functionally indistinguishable products even without reproducing program text.

Second, we have observed that the attempt to protect the valuable know-how in software has led to cycles of under- and overprotection. Those cycles are both empirically observable in the history of software case law[227] and almost inevitable given the dual nature of software as a technology and a writing, which confounds the basic assumptions of existing legal regimes.[228]

The status quo may seem comfortable, but it is unstable. Much of what is valuable in software cannot be appropriately protected by existing

223. See Greg Aharonian, <srctran@world.std.com>, Software Patenting "Crisis" Worsening, Internet Pat. News Serv. Apr. 18, 1994, URL:http://sunsite.unc.edu/patents/intropat.html (archived stories) (estimating that 11,000 patents had issued from early 1970s to early 1990s, that 1146 software patents had issued in first quarter of 1994, and that in 1994–95, 11,000 will issue). Aharonian has elsewhere estimated that IBM holds 12% of all issued software patents; other hardware companies hold many of the rest. See Gregory Aharonian, ACM Forum: Setting the Record Straight on Patents, Comm. ACM, Jan. 1993, at 17, 17. This influx of patents may partly be due to concerns that copyright alone may not provide extensive protection to software. See Mitch Betts, Vendors Seek Patents as Copyright Suits Grow, Computerworld, Oct. 11, 1993, at 109.

224. The Patent and Trademark Office has undertaken to hire new examiners with training in computer science and to develop a better classifications system for software. It has also been working with the Software Patent Institute at the University of Michigan which is building a prior art database for software. Yet, serious questions remain about its capacity to recover from several decades of deficit in keeping current about the state of the art in this field. If courts continue to construe the scope of copyright in programs constrictively, the Patent and Trademark Office may experience an even greater influx of patent applications for software-related innovations. This may make it difficult for the office to implement all needed reforms.

225. See supra notes 135–136 and accompanying text.

226. See supra notes 175–189 and accompanying text.

227. See supra notes 194–220 and accompanying text.

228. See supra notes 190–192 and accompanying text.

legal regimes, yet this very fact is what drives cycles of under- and overprotection. Left unaddressed, these cycles will not simply die down. Indeed, they are likely to grow worse as the industry matures.

The OTA has suggested that there may be a point at which it would become preferable to develop a complementary or substitute legal regime to protect computer program innovations because certain aspects of programs simply cannot be successfully accommodated within existing legal regimes. We concur: It would be far better to recognize the dual nature of software and tailor a regime to protect it, than to attempt a force-fit with existing legal regimes and risk distorting the principles on which those regimes are founded.

3 Rationale for a Market-Oriented Approach to the Legal Protection of Program Innovations

In the preceding Sections, we showed that existing legal regimes are not well-suited to protecting software innovation. They either cannot protect the principal source of value in programs—behavior—or cannot protect the incremental innovation typical in the field. Trade secrecy cannot protect what is not secret; copyright protects only text and text is largely independent of behavior; and incremental innovation cannot meet patent standards. Attempts to stretch the bounds of existing regimes to protect the incremental innovation in software will result in either too much or too little legal protection.

In response to these problems, we propose a two-part solution: (1) a protection scheme organized around the sources of value in software, namely, behavior and the applied know-how that produces it; and (2) a protection scheme grounded in principles of market economics and market preservation. In this Section, we describe what we mean by "market-oriented" and argue for the appropriateness of our approach. In Section 4, we will discuss important characteristics of the software market to which a market-oriented legal regime should be attentive. In Section 5, we will discuss three factors that can be used to predict when market failure is likely to occur as second comers appropriate innovations from a preexisting program.

3.1 Genesis of a Market-Oriented Approach

We propose taking a market-oriented approach to protecting software innovation. By this, we mean a legal regime that would not restrict appropriations of compiled know-how in programs any further than is necessary to avoid market failure and restore the kind of healthy competition that occurs when innovators enjoy sufficient natural lead time.[229] This approach follows naturally from our argument that incremental technical innovation needs protection against trivial acquisitions of equiv-

229. See Reichman, Legal Hybrids, supra note 100. Of course, there may be other market failures that no legal protection regime can rectify.

alence that have market-destructive effects.[230] Firms that can rapidly replicate the innovative behavior of another firm's product at much lower development costs can supply a market substitute that will deny the innovator an adequate opportunity to recoup its research and development expenses and make sufficient profits to justify its initial investment in innovation.[231] The law should, therefore, intervene only to the extent necessary to avert the market failure that would result from this sort of appropriation.[232] Several factors contribute to our view.

3.1.1 Incremental Innovation Has Generally Been Left to the Rigors of Free Competition

In market economies, competition is the general rule to which patent and copyright are exceptions.[233] That is, unless patent or copyright law protects an innovation embodied in a publicly distributed product, competitive imitation of it is not only lawful, but desirable from the standpoint of consumer welfare.[234] The public benefits from competitive imi-

230. For a more general discussion of the potential for market failure arising from appropriations of information innovation, see Gordon, supra note 101, at 854–59; Wendy J. Gordon, Property and Tort Responses to Failures in Markets for Intangibles 1–6 (July 16, 1994) (unpublished manuscript, on file with the Columbia Law Review). The second article has been translated into German and is scheduled for publication as Wendy J. Gordon, Systemische und fallbezogene Lösungsansätze für Marktversagen bei Immaterialgütern (L. Haberfellnertrans.), *in* Ökonomische Analyse des gewerblichen Rechtsschutzes 327–70 (Claus Ott & Hans-Bernd Schäfer eds., forthcoming 1994).

231. Because development [of information products such as software] is so costly, the difference in price between what the free rider can charge and what the developer must charge to recoup his costs is substantial. This gap in price will tend to drive consumers to purchase the goods of the free rider. To make matters worse, many of these products change rapidly . . . . Accordingly, developers cannot bring their selling prices closer to those of their competitors by extending the period during which they can recoup their costs.

Dreyfuss, supra note 6, at 901. See also Gordon, supra note 101, at 859–68 (discussing Gordon's model of the intellectual property prisoners' dilemma).

232. Professor Menell has observed that "[t]he dilemma in designing legal protection for application program code is finding the correct balance between providing programmers with sufficient lead time to exploit their programs in the market before imitators appear, and not inhibiting the rapid sequential process driving the technological advancement of application programming." Menell, supra note 1, at 1080.

233. Until very recently, United States law regarded patent and copyright as the only exceptions to the general rule of competition. Other countries, however, have been experimenting with special protection regimes for specific kinds of appropriation. See Reichman, Legal Hybrids, supra note 100, at 2453–2500; Jerome H. Reichman, Legal Hybrids Between the Patent and Copyright Paradigms, *in* Information Law Towards the 21st Century 325 (William. F. Korthab Altes et al. eds., 1992). Patent and copyright law are widely seen as legal devices that correct for market failures that would occur if all innovation revealed in publicly distributed products could be freely appropriated. See, e.g., Gordon, supra note 101, at 854–55.

234. See, e.g., Bonito Boats, Inc. v. Thunder Craft Boats, Inc., 489 U.S. 141, 142 (1989) (holding Florida statute protecting boat design preempted by federal law); Sears, Roebuck & Co. v. Stiffel Co., 376 U.S. 225, 231–33 (1964) (holding Illinois unfair competition law protecting unpatentable innovations preempted by federal law).

tation because it results in the production of more goods at lower prices as more efficient producers enter the market.[235]

Although incremental technical innovation has generally been left to the rigors of free competition, the rationale for doing so has been, apparently, the assumption that incremental innovators would have some natural lead time after introducing an innovative product to the market, during which they could charge monopoly prices.[236] Such lead time arose, in part, because manufacturers of industrial products embodying incremental innovations were able to keep secret much of the know-how required to make their products.[237] Allowing third parties to reverse engineer and appropriate incremental innovation also substantially contributed to the cumulative innovation process because those who reverse engineer often introduce improvements as they reimplement another's innovation or learn how to produce the product more efficiently.[238] Even in the face of such competition, firms with lead time generally receive enough return on their investment to provide adequate incentives for incremental innovation.[239]

In contrast to this traditional scenario, information products, such as computer software, bear so much of the technical know-how required to make them on or near the surface of the product that natural lead time for this kind of industrial product may not suffice.[240] A new exception to

235. See, e.g, Stanley M. Besen & Leo J. Raskind, An Introduction to the Law and Economics of Intellectual Property, 5 J. Econ. Persp. 3, 5–6 (1991).

236. See, e.g., Reichman, Legal Hybrids, supra note 100, at 2506–11; see also Hanns Ullrich, Standards of Patentability for European Inventions 106 (Friedrich-Karl Beier et al. eds., 1977) (arguing that innovator has natural lead time as first to establish manufacturing and distribution); Ralph S. Brown, Design Protection: An Overview, 34 UCLA L. Rev. 1341, 1388 (1987) (arguing that innovators have head start which confers market advantage).

237. See Reichman, Legal Hybrids, supra note 100, at 2514. This lead time occurred because of the time it would take others to recognize the market value of the innovation, reverse engineer it, and retool to imitate it.

238. See Friedman et al., supra note 118, at 70; see also D.V. Kerns, Additional Thoughts on Reverse Engineering (Apr. 23, 1994) (unpublished comments, on file with the Columbia Law Review). Kerns observes that

> [reverse engineering] is an impetus for higher levels of innovation . . . . [It] forces an aggressive and proactive understanding of the original innovators [sic] technology . . . . [I]n the process of understanding the way in which the innovator accomplished specific tasks, the second-comer's expertise and background can be applied to accomplish the same task but with a fresh perspective. This innovation occurs because the second-comer has the advantage of a working product with demonstrated behavior characteristics along with the required specifications.
>
> I maintain that such innovation would not necessarily occur if the original innovator merely licensed the new technology to the imitator.

239. Professor Reichman has shown that a number of legal hybrid regimes have been created to deal with instances in which natural lead time, in fact, did not suffice to provide the proper incentives environment. See Reichman, Legal Hybrids, supra note 100, at 2453–2500.

240. See Dreyfuss, supra note 6, at 903–04; Reichman, Applied Know-How, supra note 5, at 659–60.

the rule of free competition is needed for the applied know-how embodied in these products. Such an exception, like copyright and patent before it, should be tailored to insure that it will not impede competition any more than is necessary to correct the market failure it addresses.[241]

3.1.2 The Reuse of Publicly Accessible Technical Know-How Is Socially Beneficial

The new exception to the general rule of free competition for which we argue would regulate the ability of second comers to reuse not only incremental innovation, but also technical know-how embodied in publicly distributed information products. Even if there are good reasons to regulate some commercial appropriations of such know-how, we recognize that there is a strong societal interest in permitting the use of publicly accessible technical information that is unprotected by patent or copyright law. The exception must therefore extend no farther than is necessary to address market failure and preserve healthy competition.

3.1.3 The Market Has Generally Produced Competition and Innovation in the Software Industry

The history of the computer software industry demonstrates that a high degree of innovation and market growth can occur in a relatively minimalist legal environment. The industry grew from a trivial sideline to a major source of economic strength in a legal environment in which software developers regarded copyright as providing protection for code only and did not expect patent protection.[242] Most of the innovation in the software industry has been cumulative and incremental and many incremental software innovators have found substantial rewards in the market.[243]

---

241. The economists who advised CONTU took a static view of the nature of innovation in the software industry and focused their study on the incentives environment that would promote investment in stand-alone programs. See Yale Braunstein et al., Nat'l Technical Info. Serv., Economics of Property Rights as Applied to Computer Software and Data Bases (1977), reprinted in 4 Copyright, Congress and Technology 1, 7–21 (Nicholas Henry ed., 1980). More recently, economists have become concerned about the dynamic character of technical innovation. They have recognized that a somewhat different incentives environment is needed to foster cumulative innovation. See generally Robert P. Merges & Richard R. Nelson, On Complex Economics of Patent Scope, 90 Colum. L. Rev. 839 (1990) (arguing that scope of patents should favor competition for innovation, rather than pioneer inventor); Suzanne Scotchmer, Standing on the Shoulders of Giants: Cumulative Research and the Patent Law, 5 J. Econ. Persp. 29 (1991) (investigating patent protection incentives for cumulative research). For an analysis of the cumulative innovation problems presented by computer software, see Menell, supra note 1, at 1079–83; Menell, supra note 6, at 1336–39; see also Finding a Balance, supra note 2, at 21–22, 187–94 (discussing ways in which economic theory and software industry have changed since CONTU's work).

242. See Samuelson, *Benson* Revisited, supra note 5, at 1133–40.

243. For example, Lotus 1-2-3 incrementally improved upon the basic concept of a spreadsheet program introduced to the market by VisiCalc, and was a bigger economic

However, factors that provide sufficient protection and lead time in the early days of an industry may not suffice as the industry matures.[244] In those early days there are fewer barriers to entry: no brand names, no entrenched market leaders, and (in software) no installed base of existing software and no significant population of trained users. With barriers to entry lower, innovation is freer to come to market. As the market matures, however, and barriers begin to emerge, innovators will find that a more potent innovation is required to capture market share: users experienced with existing software must be motivated to incur the cost of switching. More potent innovation, in turn, will require longer design periods and development, exacerbating the problem of lead time erosion.[245] As Section 4 will demonstrate, the software market is in a different stage of maturity than it was when minimalist protection was the norm, and existing legal regimes may leave some program innovations vulnerable to market-destructive appropriations.

3.2 How a Market-Oriented Approach Might Differ from a Traditional Intellectual Property Approach

The principal mechanism of traditional intellectual property regimes is the grant of a set of exclusive property rights to qualifying creators for a set period of time.[246] Patent law, for example, provides qualifying inventors with seventeen years of rights to exclude others from making, using, or selling their invention.[247] Copyright law offers authors exclusive rights to reproduce their works in copies, to distribute copies, to prepare deriva-

---

success than VisiCalc. See Christopher Barr, From VisiCalc to 1-2-3: How Much Did 1-2-3 Really Borrow from Its Predecessor, VisiCalc?, PC Mag., May 26, 1987, at 169, 169.

> 244. It may very well be that when any industry is young, capability market power on its own is able to provide a healthy environment for small firms and for the creativity of individuals. It certainly cannot do so later on. An indispensable characteristic of such an environment is that the stakes for entry are low, meaning that new firms can come into being easily, and that individuals have little difficulty in moving from employment to start up their own firms.

William Kingston, Response *in* Direct Protection, supra note 67, at 277, 284–85 [hereinafter Kingston, Response].

245. See id. at 296.

246. The distinction between "property rules" and "liability rules" is well known in American law. Property rules typically permit owners to exclude others from interfering with their interests; liability rules permit individuals to be compensated for harm arising from unauthorized interference with their legitimate interests but do not confer power to stop the interference in the first place. See Guido Calabresi & A. Douglas Melamed, Property Rules, Liability Rules, and Inalienability: One View of the Cathedral, 85 Harv. L. Rev. 1089 (1972). Traditional intellectual property regimes fit the property rule model more than the liability rule model because they rely principally on the grant of one set of exclusive rights to all qualifying innovations for one set period of time.

247. See 35 U.S.C. § 154 (1988). In most other nations, patents last for 20 years. See Harold C. Wegner, Patent Harmonization 290 (1993).

tive works, and to publicly perform or display the works for their lives plus fifty years.[248]

Patent and copyright law occasionally deviate from the exclusive rights model by providing some circumstances in which liability rules apply for certain classes of works or uses.[249] Moreover, courts in traditional intellectual property disputes sometimes withhold injunctive relief against an infringer even when no special compulsory license provision applies, thereby effectively applying a liability rule to the dispute.[250]

A market-oriented approach for the protection of incremental innovation embodied in computer programs might employ "blocking periods" during which, as in exclusive rights regimes, certain uses of an innovation

248. See 17 U.S.C. §§ 106, 302 (1988). Some of copyright law's exclusive rights are available only to certain classes of works. See 17 U.S.C. §§ 106(4), (5). Companies that qualify as authors have 75 years of exclusive rights in their works that run from the work's first publication. See 17 U.S.C. § 302(c) (1988).

249. See, e.g., 17 U.S.C. § 115 (1988); 28 U.S.C.A. § 1498 (West 1994); 35 U.S.C. § 183 (1988); see also Mark W. Lauroesch, General Compulsory Patent Licensing in the United States: Good in Theory, but Not Necessarily in Practice, 6 Santa Clara Computer & High Tech. L.J. 41, 41 (1990); Paul S. Rosenlund, Compulsory Licensing of Musical Compositions for Phonorecords Under the Copyright Act of 1976, 30 Hastings L.J. 683, 683, 693–701 (1979). A number of countries have compulsory license provisions that, although rarely invoked, seem to induce voluntary licenses. See S. Delvalle Goldsmith, Patent Protection for United States Inventions in the Principal European Countries—Existing Systems, 6 B.C. Indus. & Com. L. Rev. 533, 535 (1965); Tetsu Tanabe, Compulsory Licensing in Japanese Patent Law, 8 Int'l Rev. Indus. Prop. & Copyright L. 42, 43, 45 (1977) (describing Japanese patent law granting compulsory license only after applicant's good faith effort to obtain voluntary license). In the United States the mechanical recording compulsory license seems to induce purchase of voluntary licenses from the Harry Fox Agency (known as "Harry Fox licenses"). See Rosenlund, supra, at 688–89. Even the GATT/TRIPs Agreement permits limited exceptions to the general rule of exclusive rights. See Reichman, GATT/TRIPS, supra note 146, at 204–10.

250. See Campbell v. Acuff-Rose Music, Inc., 114 S. Ct. 1164, 1171 n.10 (1994) (suggesting that courts consider withholding injunctive relief in close cases in which defendant has exceeded bounds of fair use); Rebecca S. Eisenberg, Patents and the Progress of Science: Exclusive Rights and Experimental Use, 56 U. Chi. L. Rev. 1017, 1077 n.230 (1989) (citing examples where relief was denied because infringement furthered public interest). The United States government has sometimes pressured firms into licensing technology that might have gone unlicensed without government intervention. See Merges & Nelson, supra note 241, at 890–91 (discussing government pressure that brought about automatic cross-licensing in aircraft manufacturing industry).

The classic statement of circumstances in which it is economically efficient to impose a liability rather than a property rule is Calabresi & Melamed, supra note 246. These authors assert that the most common reason for employing a liability rule rather than a property rule is that "market valuation of the entitlement is deemed inefficient, that is, it is either unavailable or too expensive compared to a collective valuation." Id. at 1110. High transaction costs may impede licensing even when licensing might be socially desirable. See Merges & Nelson, supra note 241, at 874 ("A substantial literature documents the steep transaction costs of technology licensing, and there is indirect evidence that these costs increase when major innovations are transferred.") (footnotes omitted); id. at 874 nn.146–47 (citing sources). We suggest that high transaction costs for many-to-many licenses of software innovations may justify the use of a liability rule to protect incremental innovation embodied in programs.

would be prohibited. However, a market-oriented legal regime could more narrowly tailor blocking periods so that they provide only the degree of artificial lead time that software developers need to avoid market failure.[251] Also, a market-oriented legal regime could use blocking periods more selectively than traditional exclusive rights laws have done. It could, for example, provide a longer blocking period against reuse of an innovation by software developers operating in the same market than against reuse by developers who sought to employ the same innovation in a remote market segment.[252] It could also exempt appropriations of program innovations having no effect on the market (e.g., some research uses).[253]

A market-oriented legal regime could also employ other mechanisms to achieve its efficiency-enhancing aims, such as requiring users to acquire automatically granted, royalty-bearing licenses for certain types of uses.[254] That is, it could develop liability rules that would, for some period of time, compensate developers when second comers used their innovations.[255] These liability rules could, for example, take into account the market segment in which a second comer operated, the nature of the appropriated entity, and the degree of similarity between the two products.[256]

4 The Nature of the Software Market and Its Consequences

If we are to have a market-oriented regime, we must take account of several characteristics of software as a product and of the software market. More specifically, we must be familiar enough with the market to be able to identify when a market failure might occur and what aspects of the healthy market must be maintained under any legal regime. This Section will discuss some respects in which software has evolved as a product; changes that have occurred in the manner in which software is distributed, licensed, and otherwise protected; the manner in which competition has been affected by changes in the nature of software products and

251. See Reichman, Legal Hybrids, supra note 100, at 2547–48.

252. See infra notes 435–439 and accompanying text.

253. For a general discussion of research uses of innovations in a field in which basic and applied science are largely inseparable, see Eisenberg, supra note 250, at 1074–78 (distinguishing among different types of research uses of patented innovations; favoring exemptions for some and liability rules for others).

254. See infra notes 415–416 and accompanying text.

255. See Reichman, Legal Hybrids, supra note 100, at 2548–51. We note that Congress has sometimes imposed a liability rule for reuse of industrial compilations of applied know-how concerning unpatented technical innovations. See Federal Insecticide, Fungicide, & Rodenticide Act (FIFRA) § 3, 7 U.S.C. § 136a (1988) (providing a right of reasonable compensation for a second comer's use of a first comer's scientific data concerning the safety of a particular pesticide). The U.S. Supreme Court upheld the constitutionality of this provision in Ruckelshaus v. Monsanto Co., 467 U.S. 986, 1019–20 (1984).

256. See infra notes 435–444 and accompanying text for a discussion of these factors.

distribution systems; and the high degree of innovation in the industry which, in part, results from development of systems that interoperate.

4.1 Evolution of Software as a Product

In the beginning, hardware was the predominant source of value in the computer industry. Software was often "bundled" with hardware sales, in effect, given away with the hardware.[257] Hardware companies had incentives to provide business applications to their customers because customers valued computers because of what they could do with them. Academic researchers created much of the early innovative software, particularly operating systems and utility programs.[258] Software not produced by hardware companies tended to be custom-developed under individual development contracts. There was, at this time, almost no distinct software industry.

This changed radically with the advent of personal computers, which ushered in the era of companies devoted solely to producing and selling individual software packages. Hardware and software markets were particularly synergistic during this era: the availability of new software products, such as spreadsheet programs, increased the utility of and hence demand for computers, while the increasing number of personal computers in use rapidly expanded the size of the software market.[259]

The market for major applications (e.g., spreadsheets, word processors, personal finance programs, and the like) dominated the first decade of the personal computer era. In recent years, these markets have consolidated, as mergers and failures winnowed out former competitors. Increasing effort and innovation have been going into developing products for niche markets (such as computer-aided design or graphic design tools).

Over the years both the view of software as a product and the character of the product have thus evolved considerably, from business applications supplied with a mainframe, to applications intended for the end-user and aimed at a mass market, to the more narrowly aimed vertical markets today.

257. See, e.g., Ferguson & Morris, supra note 80, at 7.

258. Examples include time-sharing, created at MIT and Dartmouth, the Multics operating system (MIT), and the earliest computer networks such as the ARPANET.

259. See supra note 50 concerning VisiCalc as the first application that provided those who were not computer hobbyists with a business justification for purchasing a computer. Lotus 1-2-3, in turn, became the "killer application" that made the PC, in the same way that VisiCalc had made the Apple II and that Wordstar made CP/M machines. See Stephen Manes & Paul Andrews, Gates: How Microsoft's Mogul Reinvented an Industry—And Made Himself the Richest Man in America 218 (1993).

4.2 Evolution of Distribution, Licensing, and Legal Protection Practices

A corresponding evolution occurred in distribution mechanisms, licensing practices, and means of protection. Initially, software was delivered physically with a mainframe machine during the installation and acceptance process. Given the high cost and relative infrequency of the transactions involved in buying a mainframe, traditional contract and trade secrecy law provided sufficient protection for innovation. In these early days, before time sharing or networks, there was also no ambiguity in licensing: the software came (and stayed) with the physical machine, which was in turn accessible to only one user in one location.

With the advent of mass-marketed, end-user software, new distribution channels developed, including mail order and retail store sales. Given the volume involved, it became pragmatically impossible to use traditional contract law as a means of licensing or enforcing rights. The so-called shrink-wrap licenses[260] attempted to combine a contractual mechanism with mass-market distribution, and their questionable legal status was the predictable result.[261] This massive growth in distribution also forced to the forefront the issue of copyright protection for software, because copyright is well-suited to protect information products that are publicly distributed to a mass market.[262]

But technology soon stretched the traditional copyright model. Steady reductions in hardware costs over the years made personal computers ubiquitous in almost any technologically advanced organization. As a result, large companies often have hundreds or thousands of computers. These concentrations of computers, along with the development of networks that link some or all of these machines together, created a new set of challenges for software protection and licensing. New licensing arrangements have arisen, some keyed to a specific number of multiple users, others to a specific number of concurrent multiple users, still others to a site (unlimited number of copies at a single site). Others allow unlimited internal copying followed by an annual audit.[263]

260. See supra note 26 and accompanying text.

261. Early efforts to protect software technologically with "copy protection" programs failed in the marketplace because they not only made mass copying impossible, but also did not allow consumers to make back-up copies to protect their ability to use a program. Consumers preferred unprotected software and could, in any case, write or obtain programs to defeat the copy protection programs with relative ease. See, e.g., Vault Corp. v. Quaid Software Ltd., 847 F.2d 255 (5th Cir. 1988). In place of copy protection, the industry has undertaken social efforts (e.g., advertising campaigns) to inform the public about copyright law, and legal efforts (e.g., Software Publishers Association (SPA) "audits" of suspected infringers) to enforce compliance. See Francine Knowles, Business Battles Software Piracy, Chi. Sun-Times, Oct. 12, 1993, at 51 (discussing efforts of Software Publishers Association).

262. See CONTU Report, supra note 17, at 19.

263. See generally 2 L.J. Kutten, Computer Software: Protection/Liability/Law/Forms 8-1 (1994) (discussing licensing arrangements for software). At least one major

A foreseeable development is network distribution of software. Given the digital character of software and the ability to similarly digitize its documentation (whether text, pictures, video, or sound), there is, in principle, no reason to visit a software store or use the mail to obtain it. Buyers could try out, order, and receive software and documentation over networks such as the Internet.[264] This may mean a vastly expanded market reach for both vendors and consumers.

4.3 Evolution of Competition

The nature of competition in the software market changes rapidly, with no clear outcome as yet. Initially, there was little or no competition. Early entrants like VisiCalc, 1-2-3, and WordPerfect had the field almost to themselves; the biggest challenge was often simply to grow fast enough to meet the demand.

As competitors emerged, a primary basis for competition became functionality: what the program could do. Advertisements and even professional evaluations in the trade press typically offered long lists of capabilities, catering to most consumers' desire for the largest number of features per dollar spent.

As competition has intensified, the market has changed. Price wars have occurred. The increasing commoditization of familiar applications has made customer service capabilities increasingly important.[265] Even so, the picture is still unclear: opinions differ as to the relative importance of features and customer service.[266]

Although the cost of packaged software has generally fallen, measuring the true cost of acquiring software requires taking into account the cost of using a new product. An employee earning $40,000 a year costs a company approximately fifty dollars per hour. A typical mass-market software package costs between twenty and five hundred dollars. Hence an employee who has to spend more than two hours installing and learning to use a one-hundred-dollar program has already doubled the effec-

innovative licensing arrangement was motivated by the customer's observation that " 'the current copyright rules are untenable in a corporate setting.' " Rob Kelly, 1-2-3 Skiddoo, Information Wk., July 27, 1992, at 53 (quoting Sheldon Laube of Price Waterhouse). In describing a new licensing arrangement with Borland International, the customer, a businessman, observed: " 'The whole industry still was based on an individual going out and buying a shrink-wrapped package of software .... We wanted the flexibility to copy the guy next door's software ....' " Id.

264. See, e.g., Martin LaMonica, IBM To Deliver Its Software Electronically, InfoWorld, Sept. 5, 1994, at 43.

265. See Massachusetts Computer Software Council, Inc., Software Industry 1993 Business Practices Survey 10–19 (1993) (interview with software company CEOs) [hereinafter Business Practices].

266. See id.

tive cost of the program.[267] Anyone who has attempted to install and learn to use a new software package knows that two hours is often only the beginning.[268]

Thus, there is considerable value in having users familiar with one's software. There is sufficient value, in fact, that more than one software company attempting to enter a new market niche has done so by giving away its software in an attempt to lure customers.[269] Any new product that attempts to displace a competitor will, conversely, find considerable inertia in the marketplace; users are loathe to invest the time necessary to change. This base of trained users in turn reinforces the need for interoperability: new competitors must be able to handle data in the forms produced by their competitors.

With the changing nature of competition has come changes in sources of profit in the software marketplace. It is no longer true that the only source of profit is sale of the software itself. One company selling personal financial software, for example, perceives itself to be as much in the financial services business as in the software business. It prices its software very inexpensively to attract customers in what it sees as an underpenetrated market, with its profits coming, in significant part, from the auxiliary financial services such as check printing and bill paying.[270] We expect additional examples of this phenomenon. In other cases, the initial software sale provides no profit; the goal is to train customers on the current version of the vendor's program so that they will purchase successive versions in the future.[271]

Consumers are also becoming less tolerant of stand-alone, idiosyncratic software. Hence, the market is increasingly driven by the need for interoperability, in several senses: Software must operate on a variety of hardware platforms, across networks, and most of all, programs must work together. This latter point is exemplified by the popularity of

267. See Manes & Andrews, supra note 259, at 394 (citing Microsoft president as estimating cost of training and support for software as about twice that of purchasing the software).

268. This is one place in which network externalities arise in the software context. Strong intellectual property rights may impede a socially desirable degree of standardization. See Menell, supra note 6, at 1341–45; Joseph Farrell, Standardization and Intellectual Property, 30 Jurimetrics J. 35, 36–38, 45–46 (1989) (discussing network effects).

269. For example, Computer Associates, the second largest software company, gave away more than one million copies of its personal finance program "Simply Money" in two months, and four hundred thousand copies of its "Simply Tax" program in three months for this reason. Telephone Interview between Randall Davis and Robert Gordon of Computer Associates (Mar. 17, 1994). See generally H.J. Cummins, Computer Associates Targets At-Home PC User, Newsday (Nassau & Suffolk ed.), Aug. 26, 1994, at A61, available in Westlaw, Newsday database (discussing Computer Associates' giveaway strategy).

270. Telephone Interview between Randall Davis and William Lane, Chief Financial Officer, Intuit, Inc. (July 29, 1994). Intuit developed the Quicken software product.

271. This accounts for Computer Associates' strategy with respect to giveaways of the two programs discussed supra note 269.

software "suites" containing word processors, spreadsheets, electronic mail handlers, and presentation graphics, all of which work together smoothly.

Additional evidence of the significance of interoperability is provided by the remarkable number of instances in which otherwise dogged competitors have come to recognize that their markets will be enhanced if they cooperate to develop common standards that provide interoperability.[272] Interoperability applies to data as well. Developers of commercial word processors devote significant amounts of code to ensuring that their products can read files produced by their competitors.

4.4 Character of the Industry

The industry is currently characterized by a rapid rate of innovation. New software products have been appearing recently at the rate of more than two per day, a rate that has been increasing over the past few years.[273] As we noted above, however, innovation in the software market is primarily incremental. Invention in the formal sense of the term is rare.[274]

The major market is in widespread distribution of packaged software for the personal computer: over the past several years software companies have reported that a consistent seventy percent of their revenue has been related to personal computers, with the remaining thirty percent divided up among workstations, minicomputers, and mainframes.[275] Where the early years of the computer era were marked by individual sales of software packages costing tens or hundreds of thousands of dollars, the current era is one of mass-merchandised packages costing tens to hundreds of dollars.

The market is also maturing and barriers to entry are developing. One sign of maturation is the growing importance of a base of trained users. This in itself is becoming a barrier to new entry to major application product domains. Additional barriers come from the relative stability of major products and the emergence of a few big players in the major product categories. There has been significant consolidation in these product categories and there are now a handful of players where once there were dozens.

One part of the market—operating systems—is beginning to have increasing influence. Operating systems are the foundational layer on

272. This trend is reflected in the formation of "open systems" associations. See Andrew Pollack, Computer 'Gangs' Stake Out Turf, N.Y. Times, Dec. 13, 1988, at D1, D15 (describing members of Open Software Foundation (including IBM, Digital, and HP), as well as X/Open and Unix International).

273. See Business Practices, supra note 265, at 89 (reporting that 929 software companies surveyed brought 736 new products to market in 1991, and 853 in 1992).

274. See supra notes 67–74 and accompanying text.

275. See Business Practices, supra note 265, at "Detailed Results" question 54 (unnumbered page).

which applications programs must run, and with which they must be compatible. That foundation is beginning to absorb and provide ever more functionality.[276] In recent years, for example, the widespread utility of and acceptance of graphical user interfaces led to their migration into operating systems, such as Microsoft's Windows.

Notwithstanding the dominance of a few firms in major software product markets, the market for software more generally is characterized by a large number of small companies. In a 1993 survey of a thousand software companies, almost half (forty-six percent) reported annual revenues of one million dollars or less, and fully seventy-seven percent reported revenues of five million dollars or less. Of those thousand companies, more than half—fifty-seven percent—had fifteen or fewer employees; seventy-one percent had thirty or fewer employees.[277]

Finally, the software industry is characterized by a phenomenon that may be unique among large-scale commercial marketplaces, for it includes a large body of public domain software and "shareware."[278] This software was written by people in many countries and shared via the Internet and electronic bulletin boards around the world, and ranges in size and functionality from small to quite large-scale programs.[279] The code is typically made available in one of two ways: either by putting it into the public domain, without restriction as to use, or by distributing it as shareware. Shareware is software available for the taking from bulletin boards, sometimes with the request for a modest registration fee in the range of five to twenty-five dollars. Payment is on the honor system. Registration may be attractive because it offers benefits such as periodic code updates, a printed manual, or at times, a more fully functional version of the program, but the basic spirit is of sharing interesting, enjoyable, or useful code, primarily for the sake of enjoying good craftsmanship rather than profit. Shareware is, not surprisingly, often created by individuals, but larger efforts also exist, in which packages grow quite large and sophisticated over the years as a consequence of work by many people.

276. See, e.g., John Markoff, Microsoft's Future Barely Limited, N.Y. Times, July 18, 1994, at D1.

277. See Business Practices, supra note 265, at 87 (employee data); id. at "Detailed Results" question 53 (unnumbered page) (revenue figures).

278. See Craig T. Turkington, A Comparison of Shareware and Other Software Licensing: Role of the New Copyright Office Shareware Registry, 1 U. Balt. Intell. Prop. L.J. 76 (1992).

279. As of November 18, 1994, one large electronic bulletin board called SimTel contained 11,627 programs spread across 200 different categories. On average, a dozen new files are added every day, some of which are upgrades to existing files, and some of which are new additions. Electronic Mail from Jeff Marraccini <jeff@oak.oakland.edu>, System Administrator of the OAK Software Repository, oak.oakland.edu, primary mirror site for SimTel, to David B. Oshinsky, staff, Columbia Law Review (Nov. 20, 1994) (on file with the Columbia Law Review).

4.5 Implications

The rapid innovation and strong competition is evidence that the software market is vibrant and successful. Any software protection scheme ought to take account of this, and attempt to do little more than repair market failure where and when it does occur.

The desire for interoperability means that there are natural incentives for many companies to share information (and for some, to monopolize such information[280]) and that consumers have an interest in seeing that information shared. Any legal regime should both take advantage of the existing incentive and facilitate interoperation.

The large number of small companies illustrates that the initial creation of a new software product is relatively low in capital expense and affirms the folk wisdom that innovation in this industry is often accomplished by small teams. This, in turn, suggests that any legal regime should be wary of its impact on small companies, with their limited resources.

Finally, the maturation of the market and its barriers to entry suggest that any legal regime must be able to adapt to market change. The needs early on will clearly differ from those that emerge as a market matures, and a regime that cannot adapt risks becoming irrelevant at best, an impediment at worst.

5 Primary Dimensions of a Market-Based Legal Regime for the Protection of Software Innovations

5.1 Overview of the Primary Dimensions

A market-oriented legal regime for protection of the applied know-how arising from behavior needs criteria by which to assess when market failure is likely to occur. We recommend consideration of three primary factors as the basis for making such an assessment: (1) the nature and size of the software entity or component that has been imitated; (2) the means by which a second comer obtains access to such an entity and the degree of dependence (or independence) of a second comer's creation; and (3) the degree of similarity between the products and markets of the original and second comers.[281]

Market failure is likely if the quantum and significance of the entity taken is substantial, the second comer's development effort is rapid, easy, and highly dependent on the first comer's product, the degree of similarity in the resulting products approaches identicality, and the second comer's market is proximate to that in which the first comer operates.

280. Sega and Nintendo, for instance, have attempted to protect the information necessary to interface with their videogame consoles. See Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1514 (9th Cir. 1992); Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832, 836 (Fed. Cir. 1992).

281. See Gordon, supra note 101, at 258–65 (discussing proposed tort of malcompetitive copying that resembles, to some degree, our approach).

The converse is also true: the smaller the taking, the less dependent the creation, the less similar the results, and the less proximate the markets, the less likely it is that a second comer's borrowing will undermine the first comer's incentives to invest in innovation.

This Section discusses these considerations and the potential for market-destructive effects from certain conduct. It establishes a basis upon which to construct a new legal regime for the protection of program innovations, although more empirical investigation and much debate will be needed to develop specific proposals for implementation of a new legal regime.

5.1.1 Overview of the Entity Dimension

We distinguish among five software entities, ordered below from largest to smallest in relation to the software product as a whole:

1. Program code (as a behavioral entity)
2. Program compilation as a whole (more abstract representation of behavior or design for producing behavior)
3. Subcompilations
4. Algorithms
5. Features

The degree of legal protection required to avoid market failure corresponds to the extensiveness of the taking in relation to the program as a whole. Copying code is problematic because it allows rapid appropriation of the entire software product with no expenditure of effort. Copying code is highly destructive of the developer's opportunity to benefit from its contribution to the market.

Copying of program behavior or the industrial design for bringing about behavior is also problematic because a second comer can produce a market substitute with relatively trivial effort. The appropriation of this sort of program compilation is, nonetheless, less likely to have market-destructive effects than code copying because it requires some expenditure of effort, even though far less than developing the program in the first place.

Copying of a program subcompilation, such as a portion of a program's behavior, is generally less problematic than copying the program compilation as a whole because a second comer that copies only some of a program's behavior may not, on that basis alone, offer a market substitute without expending considerable effort to design other aspects of its program. Still, subcompilations, too, may be in need of some legal protection insofar as innovators do not, in fact, have enough opportunity to earn a return that justifies investment in innovation. Although typically perceived as individual components of programs, algorithms and features may also be in need of some protection, although this is more controversial. Still, it is generally true that a competitor's appropriation of a feature is less likely to have a destructive impact on the market for the software than if the second comer appropriated code or the program compilation as a whole.

5.1.2 Overview of the Means of Access Dimension

Software development typically involves both independent and dependent creation. Programmers generally write programs without looking at the source code of other programs. However, they often make use of well-known program elements, as do engineers in other disciplines do.

Existing legal regimes tend to make a sharp distinction between dependent and independent creation,[282] but tend not to differentiate among different kinds of dependent creation.[283] We have identified several types of dependent creation in software development that are distinguishable by the means through which a second comer gains access to the know-how in the originator's program. Means of access can affect the difficulty of imitating an innovation, and hence the speed, cost, and market effect of dependent creation. We suggest the following as usefully distinguished varieties of access:

1. Dependent creation by means of:
   (a) exact duplication of code
   (b) abuse of access to nonpublic information
   (c) decompilation
   (d) extensive black-box testing, i.e., making a detailed study of the program as its instructions are executed under various conditions
   (e) other observations about the program or its functionality
2. Independent creation (no access or influence)

The right problem to worry about is not dependent creation as such, but situations in which dependent creation threatens to bring about market failure. Instead of trying to stop dependent creation in software, the

282. Independent creation of a substantially similar work does not infringe a copyright, whereas dependently creating a substantially similar work is likely to infringe. See, e.g., Alfred Bell & Co. v. Catalda Fine Arts, Inc., 191 F.2d 99, 103 (2d Cir. 1951). Independent and dependent creation are also important concepts in trade secrecy law. Independent creation of the same innovation is lawful. It is only when a firm has dependently created a product embodying the plaintiff's innovation that misappropriation of trade secrets will be found. See, e.g, Kewanee Oil Co. v. Bicron Corp., 416 U.S. 470 (1974). Patent law, by contrast, does not distinguish, for liability purposes, between dependent and independent creation. Any use of the invention will infringe patent rights. See 35 U.S.C. § 154 (1988). Dependent creation may, however, be considered as a factor in determining relief in patent infringement cases, i.e., an independent creator would not be a willful infringer subject to enhanced damages awards. See S.C. Johnson & Son v. Carter-Wallace, Inc., 781 F.2d 198, 201–02 (Fed. Cir. 1986) (holding district court's refusal to grant enhanced damages abuse of discretion upon finding of willful infringement).

283. Copyright law, for example, sometimes conflates different kinds of independent creation situations. See, e.g., Goldstein, supra note 298, at § 7.2.2 (stating that defendant can rebut inference of copying by attacking access and similarity, testifying as to independent "mental processes," or by reputational evidence).

law should regulate how rapidly certain kinds of dependently created products can be introduced to the market and under what terms.

5.1.3 Overview of the Similarity Dimension

Another important consideration in judging the potential for market-destructive effects pertains to the degree of similarity between the first and second comers' products and the markets in which they operate. The following kinds of similarities have distinguishable market impacts:

1. Exact duplication of code
2. Clones and near-clones of internal or external compilations
3. Partial clones (clones of subcompilations)
4. Substantially similar program compilations
   (a) without improvements
   (b) with improvements
5. Substantially different program compilations
   (a) migration of program elements to different markets
   (b) interoperating programs
   (c) add-on programs
6. Programs having the same general functionality but different particularized functionality

Exact duplications of code have a high potential for market-destructive effects. Clones and near-clones have a lesser potential for market-destructive effects than duplications of code, but there may still be a need to regulate how quickly they can enter the market.[284] In general, the less extensive the similarity and the less proximate the markets of the producers, the lower is the potential for market-destructive effects, and hence, the less need exists for legal regulation. When the second comer's program has the same general functionality, but different particularized functionality, there is no potential for market failure because if this kind of second comer succeeds in taking away customers from the developer of a similar product, it will be by virtue of having introduced a more desirable product, rather than because the firm appropriated the fruits of the first comer's research and development expenses.

284. We have chosen to use the term "clone" in a more expansive manner than has been common in technical parlance, where it has tended to be used to refer to a program that precisely imitates the behavior of another. See, e.g., Winn L. Rosch, The Copyright Law on Trial, PC Mag., May 26, 1987, at 157 (speaking of VP-Planner as a "near-perfect clone" of 1-2-3). One expansion is our distinction of clones, near-clones, and partial clones. See infra notes 353–367 and accompanying text for examples of clones, near-clones, and partial clones. A second expansion is our use of the term "clone" to describe a program whose internal design is substantially identical to another program's internal design. As previously demonstrated, both internal and external program compilations are industrial compilations of applied know-how that need some protection against substantially identical copying that can undercut natural lead time. See supra notes 76–110 and accompanying text. The remainder of the Article will use the term in this broader fashion.

5.2 Commentary on the Entity Dimension

5.2.1 Program Code as a Behaving Entity

The code of a program, whether in source or object form,[285] is unquestionably a valuable aspect of the program that a market-oriented legal regime should protect. This aspect of programs is now protected by copyright law in virtually every nation.[286] There are, on balance, more reasons to retain this choice than to change it, although the current copyright duration is probably longer than is necessary to protect software developers from market failure.[287]

There are several reasons why program code should be protected for a longer period than more abstract elements of programs. First, program code embodies all of the valuable behavior of the program. Unlike more abstract representations of the program, imitators can take the code wholesale with no effort. Second, developers must make substantial investments not only to design, implement, and code a program, but also to test, debug, and maintain the code.[288] Code distributed in the market reflects a substantial investment in testing and debugging beyond that required for writing the initial source code. Third, protecting code for a long time will not unduly impede competitive development of programs because a second comer can write a noninfringing program to produce the same behavior.

Although copyright in this respect works well for protecting object code, it is worth noting that the real problem with exact copying of object code is not that the prose constituting the copyrighted text has been plagiarized, but that the copyist has acquired behavioral equivalence at no cost and with no independent development effort. From an economic standpoint, the copying of code presents the most serious danger of market failure, because it undermines opportunities for the program developer to recoup its considerable research and development costs. The second comer, having essentially no development costs, can undercut the first developer's price.[289] Consumers might find this lower price attractive in the short run. If the situation is not corrected by the law or other-

285. See supra note 16.

286. Inclusion of a provision on copyright protection for computer programs in the GATT/TRIPs Agreement elevates this choice to the status of a near-universal standard of international trade law. See GATT/TRIPs, supra note 8, at art. 10, 33 I.L.M. at 287.

287. See Menell, supra note 6, at 1371.

288. See, e.g, Frank, supra note 84, at 22 (asserting that refinement and maintenance phase of software development cycle constitutes 67% of development effort).

289. This rationale led CONTU to propose copyright protection for software. See CONTU Report, supra note 17, at 10–11. At the April 1994 workshop discussing this Article, Professor Joseph Strauss of the Max Planck Institute observed that software developers must recoup the costs of developing the content of the works they publish. By contrast, "development costs" for most textual works are borne by authors who may or may not be able to recoup them by publication. See Joseph Strauss, Remarks at Symposium: Toward a Third Intellectual Property Paradigm at Columbia University School of Law (Apr. 22, 1994). An author's ability to recoup her costs will not generally be of concern to

wise, consumers suffer in the long run when software products are underproduced because optimal investments for innovation cannot be economically justified.[290]

5.2.2 Program Compilations and Subcompilations

The compilation and subcompilation concepts will be discussed in detail in the commentary on the similarity dimension.[291] The extent of protection these entities need depends largely on the extensiveness of the appropriation in relation to the products as a whole, how easy or difficult the appropriation was, how quickly a new product embodying the appropriated innovation can enter the market, the degree of similarity between the compilations embodied in the two products, the prices for which they may be offered, and the proximity of the markets of the two products.

5.2.3 Algorithms

Although algorithms are typically perceived as individual components of a software design,[292] they have a number of component parts which are organized into a coherent whole. Moreover, algorithms, conceptually speaking, determine the behavior of the programs that embody them.[293] Viewed in this way, even individual algorithms seem to be among the aspects of programs that could be protected within an industrial compilation framework.

Algorithms can be among the most valuable of the abstract elements embodied in programs.[294] An algorithm that, for example, enhances the speed at which certain program functions can be executed may provide the program embodying it with a distinct advantage over competitors.[295] To the extent that algorithm innovation is vulnerable to quick and easy appropriation, a limited artificial lead time to the developer of an innova-

her publisher, whose main interest is to make a return after covering editing, printing, and distribution costs.

290. See supra note 102 and accompanying text.

291. See infra notes 353–382 and accompanying text.

292. See supra note 37 for a definition of algorithm. The industrial compilation concept would provide a basis for protecting against commercial uses of, for example, the same six algorithms for a program of a certain type for a limited period of time. This subsection addresses the issue of whether the copying of a single, unpatented algorithm should be regulated.

293. See Newell, supra note 6, at 1032.

294. See Remarks of Robert W. Lucky, Vice President of the Applied Research Group at Bellcore, *in* Global Dimensions, supra note 6, at 385 ("[T]he real essence of software is the algorithms . . . .").

295. The Karmarkar linear programming algorithm, for example, gave AT&T an advantage in the market for flight scheduling software. See William G. Wild, Jr. & Otis Port, The Startling Discovery Bell Labs Kept in the Shadows, Bus. Wk., Sept. 21, 1987, at 69 (describing Bell Labs' Strategy of keeping Karmarkar algorithm secret to obtain "a head start at developing commercial products").

tive algorithm may stimulate a more optimal level of investment for research and development than might otherwise occur.

Notwithstanding Supreme Court decisions holding that program algorithms are unpatentable,[296] a considerable number of patents have issued in recent years to developers of new program algorithms.[297] An important social benefit of algorithm patents is that they may reveal any breakthrough discovery to the public, so that societal resources are not spent redundantly to rediscover it. Furthermore, holders of algorithm patents will often have incentives to license use of their algorithms by others, particularly in application domains remote from the market in which the patent holder's own products may operate. This has the market-preserving effect of getting compensation to innovators for the value they introduce into the market, while allowing second comers to develop innovative new applications for the original discovery.

Yet, patents may not be a market-preserving form of legal protection for algorithms in all respects. Earlier Sections have discussed our reasons for believing that the needs of the software industry and the protection mechanisms of the patent regime are mismatched.[298] Many of these apply to algorithms as well as to other kinds of software innovations. An additional consideration that disfavors patents for algorithms is the effect that algorithm patents may have on the fields of computer science and mathematics.[299] Algorithms *are* abstract mathematical objects that determine how certain inputs will be transformed into certain outputs. They are, or very nearly are, laws of nature and mathematical truths that patent and copyright law have generally regarded as better left unprotected by law. The field of computer science has advanced in part because researchers in the field have been free to experiment with algorithms.[300]

Consequently, a market-oriented legal regime that protected algorithms would regulate only commercial implementations of algorithms, whereas patent law regulates virtually all uses of inventions.[301] It would preserve the trade secrecy option for algorithms,[302] but might also en-

296. See supra notes 123–128 and accompanying text.

297. See supra notes 129–130 and accompanying text.

298. See supra notes 121–140 and accompanying text.

299. See Newell, supra note 6, at 1024–28.

300. See id. at 1026–33; Mathematical Report, supra note 221, at 3–4.

301. See 35 U.S.C. § 154 (1988) (granting exclusive rights to use an invention, not just to sell products embodying it); see also Chisum, Algorithms, supra note 1, at 1017–18 (expressing concern that patent law might impede scientific and research uses of algorithms). For a more general discussion of patent law and the desirability of permitting scientific and research uses of inventions in fields in which basic and applied research are closely related, see Eisenberg, supra note 250.

302. Trade secrecy is the principal way that software developers today protect algorithms from market-destructive appropriations. When an algorithm is embodied in program code, its know-how is not borne on the surface of the product, although it may be discovered by decompilation. Protecting an algorithm by trade secrecy will not necessarily afford its developer a permanent advantage in the market. Other developers may independently come up with the same algorithm or may be able to infer the algorithm by

courage disclosure by providing a registration system through which developers of innovative algorithms could seek a period of blockage or compensation for use of their innovations.[303] This would likely lessen the pressure that in recent years has led more software developers to seek patents for algorithms.

5.2.4 Features

Features, like algorithms, tend to be perceived as individual components of software products. However, features are often composites of elements. Some are substantial in size and complexity, such as the macro feature of Lotus 1-2-3.[304] Other features are more modest in size, such as the "zooming" animation produced when Macintosh files are opened up.[305] Some features, however, consist of a single element.[306] Features that consist of only one or a small number of elements are probably too small in relation to the software product as a whole to affect investment incentives. Hence, they should probably be exempt from regulation by a market-oriented legal regime. Complex features should be regarded as subcompilations and protected against market-destructive appropriations.

Software developers have often adopted features that were first introduced in another firm's program. This adoption sometimes occurs in the same market as the product that introduced the feature,[307] and other times, in an adjacent market.[308] But features initially embodied in one kind of program sometimes "migrate" to remote markets.[309] There has been relatively little objection to feature copying or migration among software developers, at least as long as the second adopter independently

analyzing the embodying program's behavior. In the meantime, however, trade secrecy protection for algorithms will tend to give developers lead time approximately covering the expense and risk of creating their innovations. If an algorithm is obvious, other developers are likely to figure it out relatively quickly and reimplement it in their own code if the algorithm is superior to what they previously used. The less obvious an algorithm is, the longer it will likely take competitors to figure it out, and the greater will be the natural lead time advantage to the innovator. Either way, the innovator is likely to have some natural lead time in the market.

303. See infra notes 429–434 and accompanying text.

304. See Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 64–65 (D. Mass. 1990) (describing Lotus macro feature).

305. See Apple Computer, Inc. v. Microsoft Corp., 799 F. Supp. 1006, 1036–37 (N.D. Cal. 1992) (discussing Apple's copyright claims for "zooming" animation).

306. An example of a primitive feature is giving users a choice between white or blue as a background color for their screens.

307. Most commercially distributed word processing programs have adopted features from one another.

308. The Sideways program, discussed infra notes 396 and 438 and accompanying text, started out as a feature in an adjacent market, but ended in Lotus 1-2-3 and, therefore, in its market.

309. The tool bar (i.e., a bar of menu command options found in many applications with graphical user interfaces) is an example of a feature that has migrated to a wide variety of application domains.

implemented the feature (i.e., wrote its own code).[310] This is noteworthy given the frequency with which feature copying has occurred.

It is easy to understand why there would be little or no objection to feature migration to very different application domains. If a program does not compete in the same market as the program from which the feature was copied, it is unlikely that the innovator will perceive the feature migration as injurious. Even when features have been copied by firms operating in the originator's marketplace, however, objections have tended to center on the copying of all or a substantial subset of features rather than on individual features as such.[311]

A plausible explanation for the general acceptability of copying individual features among software developers is that there has not as yet been any serious market failure resulting from such copying. The firm that introduces a valuable feature to the market will have some natural lead time protection by virtue of being first. It may take a year or so for others to introduce a competing product with the same feature into the market. While others are busy copying the feature to introduce it into later versions of their own programs or into new programs, the originator may be refining it or adding other new features to its products. These refinements or new features may give the developer a new lead time advantage over those whose development efforts have been focused on catching up to the previous innovation.

Whether individual features need artificial lead time protection, other than that which might be available for those complex enough to be protectable subcompilations, may depend on whether the pace of software development speeds up. One sign that developers are becoming concerned about feature copying as a prospective source of market failure is that recent software copyright lawsuits have focused on the copying of features as a basis for infringement claims.[312] The availability of a legal regime that furnishes a relatively self-executing means to provide some market-oriented protection to features may, as with algorithms, alleviate some of the pressure to use copyright or patent law for this purpose.

310. In some cases, judges have relied on scenes a faire and related doctrines to reject infringement claims based on the copying of features. See, e.g., Brown Bag Software v. Symantec Corp., 960 F.2d 1465, 1472–73 (9th Cir.), cert. denied, 113 S. Ct. 198 (1992).

311. See, e.g., Apple Computer, Inc. v. Microsoft Corp., 799 F. Supp. 1006, 1015–16 (N.D. Cal. 1992) (claiming copying of substantial number of features).

312. See sources cited supra notes 304–305; see also Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203, 205 (D. Mass. 1992) (complaining about copying of macro feature); John Richardson Computers, Ltd. v. Flanders [1993] F.S.R. 497, 515 (Ch.) (British case complaining of copying of certain features of program for producing pharmaceutical prescriptions).

5.3 Commentary on Access Dimension: Of Dependent and Independent Creation

In a market-oriented legal regime, the lawfulness of different forms of reverse engineering and dependent creation would be assessed, in part, by whether they allowed second comers to acquire behavioral equivalence with only trivial effort in comparison with high costs of initial development. A given form would be lawful unless it would induce market failure.[313]

5.3.1 Dependent Creation and Reverse Engineering Are Normal in Technological Fields

Reverse engineering and dependent creation are well-accepted practices in all technological fields. Engineers are expected not only to know the state of the art, but also to utilize efficient solutions to technological problems that others have contrived. In the United States, the only technological breakthroughs that the law has protected against imitative copying were those that complied with the rigorous process and standards of the patent system.[314]

The standard rules about reverse engineering and imitative copying of unpatented technical innovation may need to be adjusted somewhat as applied to computer software because of the greater vulnerability of innovations embodied in software to trivial acquisition of equivalence.[315] However, it is both fruitless and socially undesirable to insist, as some software copyright proponents have done, that all software should be developed completely independently from the existing software technology base, without reference to or utilization of techniques and methods employed by others.[316] Dependent creation of software, like dependent creation in other technological fields, is desirable unless it has market-destructive consequences. The remainder of this subsection will evaluate the market impact of various forms of dependent creation of software and will judge the legitimacy of various means of reverse engineering in terms of these impacts.

313. See supra notes 102–110.

314. See, e.g., Bonito Boats, Inc. v. Thunder Craft Boats, Inc., 489 U.S. 141, 159 (1989) (holding unpatented boat hull could be copied by plugmold process). In Switzerland, unfair competition law can sometimes be used to stop "slavish imitations." See Reichman, Legal Hybrids, supra note 100, at 2474–75.

315. See supra notes 93–100 and accompanying text.

316. See, e.g., Clapes, supra note 5, at 208 (arguing that large and complex programs "exhibit all the attributes of those kinds of imaginative literary works of the type with which the general public—and the copyright law—are already quite conversant," and that requiring software developers to create software independently is necessary to promote proper incentives environment for software).

5.3.2 Reproduction of Program Code as a Form of Dependent Creation

If the general rule that unpatented technical innovations embodied in mass-marketed products could be freely appropriated were to be applied to program code, it would result in market-destructive appropriations of program innovations and underinvestment in the development of commercially valuable programs.[317] In such a legal environment, the developer of an innovative software product would have to recoup all of its research and development expenses and make a profit on the first (and likely only) sale of the product, a scenario unlikely to lead to a high degree of investment in software.[318]

The decision to extend copyright protection to program code is defensible on economic grounds. Copyright allows software developers to spread recoupment of their research and development costs over sales of multiple copies and obliges competitors to write their own program text if they want to offer a competing product.[319]

5.3.3 Abuse of Access to Documentation as a Means of Dependent Creation

Another market-destructive form of dependent creation of software occurs when the imitator abuses access to source code, flow charts, or other documentation in breach of a confidential relationship or an enforceable contractual restriction. This abuse can facilitate the trivial acquisition of functional equivalence, although it requires more effort to engage in abuse-of-confidence dependent creation than in exact copying of publicly distributed program code. Trade secrecy and contract law protect against this form of dependent creation.[320]

317. Trade secrecy alone provides no protection to mass-marketed software products. See, e.g., Videotronics, Inc. v. Bend Electronics, 564 F. Supp. 1471, 1475–76 (D. Nev. 1983) (holding mass-marketed videogame program unprotected by trade secrecy law).

318. See CONTU Report, supra note 17, at 11.

319. As defensible as this decision was on economic grounds, a copyright-like form of legal protection, such as that created for semiconductor chip designs, would have achieved the same result, at least in the domestic market. The choice of copyright for program texts has had the advantage of protecting programs internationally through existing copyright treaties and conventions. See Dworkin, supra note 6. Now that much of the international community has agreed to protect computer programs through copyright law as part of the GATT, see GATT/TRIPs, supra note 8, at art. 10.1, 33 I.L.M. at 87, copyright can provide a worldwide means for taking action against piracy of code. In truth, GATT might as easily have included a provision reflecting a consensus on another form of legal protection for programs. If a new consensus on legal protection of computer programs emerges, GATT could be amended to reflect this consensus. See supra notes 146, 249, and infra note 467 for further discussion of the GATT/TRIPs agreement.

320. See, e.g., Boeing Co. v. Sierracin Corp., 738 P.2d 665, 673 (Wash. 1987) (relying on both trade secrecy and contract law to protect unpublished information).

Because trade secrecy law is grounded in unfair competition principles,[321] courts typically enjoin a trade secret misappropriator from further abuse of access. However, courts in trade secrecy cases tend to limit injunctive relief to that which is necessary to preserve the proper competitive environment.[322] Injunctions are, for example, frequently granted for the amount of time that it would have taken someone to lawfully reverse engineer the information or discover it independently.[323]

Trade secrecy remedies thus ensure that both the original innovator and the dependent creator have an opportunity to participate in the marketplace in a market-preserving way. The trade secret holder can get an injunction to restore the lead time advantage the holder would have had if no abuse of access had occurred. The dependent creator is not permanently banned from competing in the marketplace. However, its entry to the market is delayed long enough to ensure the innovator's lead time is not destroyed through abuse of access.[324]

5.3.4 Decompilation as a Means of Dependent Creation

A considerable controversy has erupted in recent years concerning decompilation of computer program object code. Strictly speaking, decompilation is not itself a form of dependent creation. Rather, it is a means of getting access to information that, once known, may become the basis of a subsequent dependently created program, i.e., a program incorporating whatever information was discerned from the decompilation process.

Thus far, much of the legal debate has concerned the application of existing legal regimes. We suggest that the focus of the debate should be on the market effects of dependent creations resulting from decompilation.

Under traditional principles of trade secrecy and patent law, purchasers have been free to disassemble a publicly distributed technological product embodying an unpatented innovation.[325] If the disassembler thereafter constructed a product embodying the innovation and sold it in competition with the original, the innovator had no legal recourse (un-

321. See Kewanee Oil Co. v. Bicron Corp., 416 U.S. 470, 481–82 (1974). The American Law Institute is in the process of preparing a new Restatement of the Law of Unfair Competition that will include provisions on trade secrecy.

322. See, e.g., Northern Petrochemical Co. v. Tomlinson, 484 F.2d 1057, 1060–61 (7th Cir. 1973) (holding owner of process adequately compensated by competitor's refraining from engaging in competition for period of time necessary to develop process independently).

323. See, e.g., Winston Research Corp. v. Minnesota Mining & Mfg. Co., 350 F.2d 134, 142 (9th Cir. 1965). See generally Note, Trade Secrets: How Long Should an Injunction Last?, 26 UCLA L. Rev. 203, 215–16 (1978) (discussing benefits of finite injunctions).

324. We emphasize this point because some of the market-preserving principles of trade secrecy law could be applied to software under a market-oriented legal hybrid regime.

325. See supra note 314 and accompanying text.

less the second comer misrepresented the source of the goods).[326] Applying this rule to software, even as amended by the copyright stricture against selling someone else's code, would result in the simple requirement that subsequent developers write their own programs to reimplement the same technical design.[327]

To overcome this seemingly obvious result, some have argued that decompilation is an infringement of copyright because it necessarily involves making intermediate copies of program code.[328] Its defenders counter that decompilation is a fair use of the copyright in a program, particularly when undertaken in order to gain access to its functional details, such as information necessary to achieve interoperability.[329] As long

326. See, e.g., Bonito Boats, Inc. v. Thunder Craft Boats, Inc., 489 U.S. 141, 149–50 (1989).

327. As we have explained supra notes 158–160 and accompanying text, this is all that we believe Congress to have intended that copyright law should protect in computer programs.

328. Those who oppose decompilation say that it cannot be a fair use of the program copyright because the copying is commercial, decompilers are trying to gain access to something which the developer has chosen to keep secret, and the whole of the program must be copied to decompile it. In addition, they argue that there is harm to the market for the copyrighted work, since those who decompile often intend to develop competitive substitutes for the programs they decompile. See, e.g., William F. Patry, The Fair Use Privilege in Copyright Law 400–02 (1985); Allen R. Grogan, Decompilation and Disassembly: Undoing Software Protection, Computer Law., Jan. 1984, at 1; see also Miller, supra note 1, at 1014–17 (criticizing court decisions permitting decompilation as fair use). Grogan argues that decompilation is wrongful under both copyright and trade secrecy law. If intermediate copying is copyright infringement, then decompilation of program code will be an improper means of obtaining the secret, giving rise to a claim of trade secret misappropriation as well as of copyright infringement. See Grogan, supra, at 9–10. The decompilation-as-infringement position is, in essence, an attempt to convert copyright into trade secrecy. Although United States courts have rejected this argument, European policymakers have decided, in order to protect investment in software, to permit this conversion to occur except for decompilation to get access to the information necessary to achieve interoperability. See EC Directive, supra note 7, Recitals, art. 6, at 42–45. For a discussion of the history of the Directive, see Thomas C. Vinje, The Legislative History of the EC Software Directive, *in* A Handbook of European Software Law 39 (Michael Lehmann & Colin Tapper eds., 1993). The Directive also restricts the use of interface information obtained by decompilation. See EC Directive, supra, at art. 6.2. Thus, publication of a book about undisclosed interface information, such as that recently published in the United States about undocumented calls of the Microsoft Windows program, would have been unlawful in Europe.

329. Those who think decompilation should be lawful argue that the decompilation is mainly undertaken for research and analysis purposes. Most copyrighted works reveal their contents to anyone seeking to read them, which makes it unnecessary to reproduce the texts to get access to the ideas, techniques, methods, and information they contain. By contrast, the unprotectable elements embodied in computer programs can only be discerned by decompilation. If decompilation is forbidden, software developers could use copyright law to get de facto monopolies on functional processes and systems embodied in programs that have not met patent standards. Although the whole of a program is generally copied during decompilation, the copy is intermediate in character. So long as the decompiler does not appropriate expression from the program, no harm cognizable by copyright law generally occurs. See Ronald S. Laurie & Stephen M. Everett, Protection of

as the decompiler refrains from copying expression from the decompiled program, they assert, no harm of concern to copyright law can occur to the market for the decompiled program.[330]

Two federal courts of appeals have decided that making intermediate copies of programs is fair use when done for a legitimate purpose, such as to gain access to information necessary to develop a program that can interoperate with other programs or with hardware.[331] While these rulings are unexceptionable as a matter of copyright and trade secrecy law,[332] they may leave software internals more vulnerable to market-destructive appropriations than is desirable.

Trade Secrets in Object Form Software: The Case for Reverse Engineering, Computer Law., July, 1984, at 1. Support for the lawfulness of decompilation has generally been strong within the legal academic community. See, e.g., Sega Amicus Brief, supra note 5; LaST Frontier Report, supra note 1, at 25; Jessica Litman, Copyright and Information Policy, Law & Contemp. Probs., Spring 1992, at 185, 196–201; Charles R. McManis, Intellectual Property Protection and Reverse Engineering of Computer Programs in the United States and the European Community, 8 High Tech. L.J. 25, 28 (1993); Reichman, Applied Know-How, supra note 5, at 700–03. But see Miller, supra note 1, at 1014–22 (arguing that decompilation should not be fair use under Copyright Act, dealing with arguments made in *Atari* and *Sega*).

Some debate exists in the United States concerning the circumstances under which developers can avoid this result by contract provisions forbidding decompilation or disassembly. See, e.g., McManis, supra, at 87; Rice, supra note 26, at 162–66, 186–87 (arguing that contractual restrictions contravene copyright policy and lack statutory foundation). In member states of the European Union, contractual restrictions on decompilation are unenforceable insofar as they attempt to avoid the decompilation-necessary-to-achieve-interoperability provisions of the software directive. See EC Directive, supra note 7, at art. 9.1; see also id. at art. 6 (decompilation).

330. Developing a noninfringing work that embodies another work's ideas, even if it is far more successful than the work whose ideas it borrows, does not harm market interests in any way that concerns copyright. See, e.g., Bevan v. Columbia Broadcasting Sys., Inc., 329 F. Supp. 601, 604–05 (S.D.N.Y. 1971) (holding that "Hogan's Heroes" program did not infringe copyright in Stalag 17 play despite numeorus similarities).

331. See Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1520–21 (9th Cir. 1992); Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832, 843 (Fed. Cir. 1992). These decisions apparently permit decompilation to gain access to other "uncopyrightable" elements of programs as well. See Pamela Samuelson, Fair Use for Computer Programs and Other Copyrightable Works in Digital Form: The Implications of *Sony, Galoob*, and *Sega*, 1 J. Intell. Prop. L. 49, 86–98 (1993).

332. It would have distorted traditional principles of intellectual property law to allow copyright to be used as trade secrecy law. Copyright has traditionally promoted the dissemination of knowledge, not its suppression. "[I]t should not be forgotten that the Framers intended copyright itself to be the engine of free expression. By establishing a marketable right to the use of one's expression, copyright supplies the economic incentive to create and disseminate ideas." Harper & Row, Publishers v. Nation Enters., 471 U.S. 539, 558 (1985); L. Ray Patterson, Free Speech, Copyright, and Fair Use, 40 Vand. L. Rev. 1, 4 (1987). As noted above, supra notes 118, 119, 314 and accompanying text, studying products sold in the marketplace has always been a fair means of acquiring know-how. It is interesting to note that the Ninth Circuit Court of Appeals in *Sega* expressed concern that Sega's argument would, in effect, cause copyright law to provide patent-like protection without meeting patent standards or procedures, see *Sega*, 977 F.2d at 1527, but did not notice that the argument would, in effect, have made copyright into a trade secrecy law.

In our view, decompilation should be regulated by the law—although not necessarily by copyright law—only if and to the extent that it permits competitors to acquire behavioral equivalence with only trivial effort and thereby induces market failure. This can occur only if, through decompilation, the imitator can create a virtually identical (and equally good) program costing significantly less than was required to develop the decompiled program.

Given the present state of technology, we doubt that decompilation presents a serious threat of market-destructive effects.[333] Decompilation tends to be undertaken when it is the only way—short of a prohibitively expensive license—to get access to essential information. As a consequence, software developers today are often able, in practice, to maintain as trade secrets much of the know-how embodied in their programs. Because reverse-engineering technology may substantially improve, however, the know-how embodied in program internals may eventually become as vulnerable to rapid appropriation as is currently true for program externals.[334]

Beneath the anxiety that some express about decompilation is a tacit recognition that the know-how embodied in internal designs of programs cannot be protected by existing legal regimes once it is discerned, however this may occur. Many of those who have seen the looming problem, however, are nonetheless trying to stop decompilation with copyright, refusing to recognize that the source of the problem is that existing legal regimes cannot protect valuable aspects of software.[335]

The attempt to outlaw decompilation is, in our view, misguided because it focuses on the wrong problem. The real problem is that the know-how embodied in programs is valuable, yet potentially vulnerable to trivial acquisitions of equivalence. If the law addresses the real problem underlying the decompilation controversy by devising a market-oriented form of legal protection for the compiled know-how and industrial designs embodied in programs, it may become unnecessary to regulate decompilation.

However, even if policymakers decide to regulate decompilation to some degree,[336] they should recognize that a complete ban is contrary to

333. Decompilation is very difficult and time-consuming to do. In addition, even after the decompiler discerns something useful from the decompilation, he or she must still expend effort to embody it in a differently expressed program. See supra notes 89–91 and accompanying text; see also Finding a Balance, supra note 2, at 146–48 (discussing difficulties inherent in decompilation process).

334. See sources cited supra note 115.

335. See, e.g., William T. Lake et al., Tampering with Fundamentals: A Critique of Proposed Changes in EC Software Protection, Computer Law., Dec. 1989, at 1, 5 ("Decompilation of a computer program does not provide an imitator with just a good start in producing a competing product; it gives him virtually everything necessary to produce a functionally identical product.") (emphasis omitted).

336. Taking a market-oriented approach to regulation of decompilation might lead to making distinctions between different purposes for decompiling. For example, the law

basic norms of competition law.[337] It would limit access to and reuse of information about incremental innovations in mass-marketed products out of proportion to the harmful effects such uses have in the marketplace. It would also result in socially wasteful replications of effort which could have other sorts of market-destructive effects.

5.3.5 Detailed Study of Program Externals as a Means of Dependent Creation

A more common, less costly, and less controversial means of engaging in dependent creation of computer programs involves use of a technique known as "black box" testing.[338] Studying a program by watching it in operation can reveal information about its internal construction.[339] Generally, however, the intent is simply to make an exhaustive list of all the things the program does, so that the imitator can replicate some or all of its functionality in an independently written program. To the extent the subsequently developed program incorporates knowledge derived from this sort of study, the second program can be said to have been dependently created. Although virtually no one seems to argue that black box testing ought to be prohibited as an illegitimate form of reverse engineering,[340] some lawsuits have challenged the lawfulness of dependently created products arising from such study.[341]

Reimplementing another program's functionality is not, in our view, a copyright, patent, or trade secret misappropriation.[342] However, it can have market-destructive effects because of the relative ease and low cost with which a skilled engineer can appropriate the know-how required to produce certain behavior by watching another firm's program in action. If the engineer thereafter makes a functionally indistinguishable program that arrives in the market very quickly, it will undercut the ability of the

might regulate decompilation used to get access to internal know-how in order to make a competing program more heavily than decompilation in pursuit of the information necessary to create a program that could interoperate with the decompiled program but did not compete with it. See infra notes 435–438 and accompanying text.

337. See sources cited supra note 233.

338. See supra notes 23–24 and accompanying text.

339. Watching a spreadsheet program in operation, for example, may allow a skilled programmer to infer the algorithm that must have been used to do recalculation.

340. In one case, Vault Corp. v. Quaid Software Ltd., 847 F.2d 255, 261 (5th Cir. 1988), a software developer argued that executing the program in order to study its operations and make a program that interacted with it was unlawful because it involved making copies of the program beyond the use-copying privilege embodied in 17 U.S.C. § 117 (1988). The Court of Appeals rejected this argument. See id.

341. In Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 69 (D. Mass. 1990), Lotus challenged Paperback's black-box reimplementation of 1-2-3. Copyright claims for infringement of "feel" are similar. To say that a software developer copied the "feel" of an original program is to say that the imitator observed the target program's external activities and reimplemented them. See Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203, 220–21 (D. Mass. 1993).

342. See supra notes 325–327 and accompanying text.

first firm to recoup its considerable research and development expenses.[343]

5.3.6 Independent Creation

Software developers also engage in a substantial amount of independent creation, both in writing code and in designing the industrial compilation of the program. They often do independent development for reasons other than the difficulty of reverse engineering. They may, for instance, prefer their own solutions to a technological problem over solutions adopted by others. Or they may be working with a different set of constraints, which makes an approach used in another software product unsuitable even for a similar project.[344]

In software, as in other technical fields, independent workers may achieve similar results. Where the compilation has a small number of elements, consists of elements that software engineers would typically include in a program of that kind, or pertains to a particularly efficient solution, similarities in program compilations may be as likely to be products of independent effort as of dependent imitation.[345]

5.4 Commentary on Similarity Dimension

The copyright case law has tended to recognize only two kinds of similarities among programs: literal similarities involving the copying of code;[346] and "nonliteral" similarities, frequently called "structure, se-

343. See supra note 102. Other activities are also potential means of dependent creation of software. Reading articles or advertisements about software products in magazines can also reveal useful know-how about a program. Watching a program being demonstrated at a conference may reveal the features it offers that would be desirable in other products as well. This sort of dependent creation is a normal part of the professional work in technological fields and is widely regarded as fair competitive behavior. See, e.g., Pamela Samuelson & Robert J. Glushko, Comparing the Views of Lawyers and User Interface Designers on the Software Copyright "Look & Feel" Lawsuits, 30 Jurimetrics J. 121, 132–33 (1989).

344. When it would be inefficient to use the same technical design, but a second comer uses it anyway, the second comer may be engaging in the sort of imitation that has market-destructive effects. On occasion, courts have regarded this kind of conduct as indicative of copyright infringement. See, e.g., E.F. Johnson Co. v. Uniden Corp. of America, 623 F. Supp. 1485, 1494 (D. Minn. 1985). Use of the same algorithm when others would have been more efficient may be indicative of market-destructive cloning, but not of copyright infringement. In this regard, we disagree with Professor Miller. See Miller, supra note 1, at 1009 n.153.

345. Courts have ignored commonplace similarities in some software copyright cases, using the scenes a faire or merger doctrines. See, e.g., Brown Bag Software v. Symantec Corp., 960 F.2d 1465, 1472–73 (9th Cir. 1992). See supra note 58 for the Second Circuit's view that use of the same highly efficient design in two programs has no bearing on proving dependent creation.

346. See, e.g., Apple Computer, Inc. v. Franklin Computer Corp., 714 F.2d 1240, 1253 (3d Cir. 1983) (copying of operating system programs), cert. dismissed, 464 U.S. 1033 (1984).

quence, and organization."[347] We have found it helpful to distinguish a broader array of similarities among programs and program elements.[348] Each has a somewhat different potential to bring about market-destructive effects.

5.4.1 The Two Easiest Cases: Similarity in Code and in General Purpose or Function

The most trivial way to acquire functional equivalence with another program is, as we have said, the exact duplication of program code.[349] Whether the identity in code is complete or only partial, code copying is an easy case which copyright forbids with no difficulty.[350] The result obtained under copyright law is consistent with the market-preserving principles we seek to promote in the regulation of competition in the software industry.

The easy cases at the opposite end of the spectrum are programs which are similar only at high levels of abstraction, such as in their general purpose or function. No one should have even a short-term monopoly for being the first to think of "computerizing" certain functions.[351] If a second comer implements these functions in a very different way, the similarities present no danger of market failure. Both competition and innovation will be enhanced by the appearance of different products in the market that implement the same or similar functions in different ways.[352]

5.4.2 Of Clones, Near-Clones, and Partial Clones

Many software developers in the mid-1980s thought that software clones would be legal because clones of hardware products were.[353] Computer clones not only expanded the supply of computers that could achieve the same functionality, but often featured incremental improvements over the hardware they cloned. They also increased price competi-

347. See, e.g., Lotus Dev. Corp. v. Paperback Software Int'l, Inc., 740 F. Supp. 37, 55-56 (D. Mass. 1990).

348. For an overview of our spectrum of similarities, see supra note 284 and accompanying text.

349. See supra note 95 and accompanying text.

350. See, e.g., Apple Computer, Inc. v. Formula Int'l, Inc., 594 F. Supp. 617, 622–23 (C.D. Cal. 1984). Courts have sometimes found reproduction of short code segments to be non-infringing when necessary, for example, to achieving interoperability. See, e.g., Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1524 n.7 (9th Cir. 1992).

351. Some patents have, unfortunately, been for high-level concepts of this sort. See, e.g., U.S. Patent No. 5,105,184, supra note 134.

352. See Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 65–67 (D. Mass. 1990) (noting differences among spreadsheet programs and competitive benefits of these differences).

353. See, e.g, Rosch, supra note 284, at 157.

tion in computer markets.[354] The interchangeability of software and hardware and the independence of program text and behavior caused many to think that software clones were and should be lawful. Software clones seemed capable of offering the same competitive benefits in software markets as clones had provided in hardware markets.[355] All one was obligated to do in preparing a software clone, under this view, was to write one's own code to reimplement the same behavior.[356]

Cloning the behavior of a program has occasionally, although wrongly in our view, been regarded as copyright infringement.[357] Yet, we have sympathy with those who argue that developers of innovative program behaviors should have an opportunity to benefit from the value they have contributed to the market before others can enter the market with competitive substitutes. It would be nice if natural lead time sufficed. But because program behavior can be so easily discerned and reimplemented by skilled programmers, some artificial lead time may be required so that innovators have an opportunity to recoup their research and development expenses, averting market failure.[358]

354. This was one of the defendants' arguments in *Paperback,* 740 F. Supp. at 77. Paperback's spreadsheet program had a database component that Lotus 1-2-3 did not have at the time Paperback's product came on the market.

355. See, e.g., Rosch, supra note 284, at 164.

356. This was essentially the defendant's argument in *Paperback,* 740 F. Supp. at 73 (proposing "bright-line" rule that copyright protects only source and object codes); see also Digital Communications Assocs., Inc. v. Softklone Distribut. Corp., 659 F. Supp. 449 (N.D. Ga. 1987) (screen displays not copyright protected because various programs could produce same screen). An interesting aspect of software copyright cases involving clones is that they have typically focused on the selection and arrangement of command terms in order to take advantage of the copyright case law on compilations in which originality in selection and arrangement of words has often been protected. See Ginsburg, supra note 74, at 1893–94. For defense efforts to characterize command hierarchies as akin to blank forms, see, e.g., *Softklone,* 659 F. Supp. at 460–62, or functional arrangements of words, see, e.g., Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203, 212–14 (D. Mass. 1992) that are unprotectable by copyright law under precedents such as *Baker v. Selden,* 101 U.S. 99 (1879) have generally not found favor in the courts, despite their plausibility. The selection and arrangement of words, dials, or switches, constituting the user interface of a physical machine, such as the flight instrument panel of an airplane, would easily be recognized as unprotectable by copyright law. The fact that many elements of the flight instrument panel have words above or beneath the dials, switches, or gauges does not change the copyright status of flight instrument panels. Command terms that serve as the means for invoking software machine functions should be viewed in the same manner. On occasion, it becomes apparent that plaintiffs in user interface cases are really trying to stop competitors from selling a functionally indistinguishable product, not just protecting a display of command terms. See Lotus Dev. Corp. v. Borland Int'l, Inc., 831 F. Supp. 223, 234–35 (D. Mass. 1993) (finding "nonliteral" infringement because Borland program reproduced the Lotus program's sequence of functions without displaying Lotus's user interface commands).

357. See *Borland,* 831 F. Supp. at 234.

358. Because of the *Whelan, Softklone, Paperback,* and *Borland* decisions, clones of software products have been deterred from the market. See Whelan Assocs. v. Jaslow Dental Lab., Inc., 797 F.2d 1222 (3d Cir. 1986) (finding infringement in part because defendant's product performed some of same functions in same way), cert. denied, 479

One of the clearest examples of a potentially market-destructive software clone was the VP-Planner program that Paperback Software sold as a "work-alike" of Lotus 1-2-3.[359] An example of a near-clone[360] is the Key Reader feature of Borland's Quattro Pro.[361] This feature allowed users familiar with the Lotus command hierarchy to utilize the same keystrokes to execute the same sequences of functions as in the Lotus program, effectively reproducing the behavior of 1-2-3 without any display of the Lotus commands.[362] The Key Reader feature permitted Borland's product to achieve behavioral equivalence with the Lotus program with only slightly more effort than if it had reproduced the command hierarchy exactly.[363] Borland's inclusion of additional commands, additional functionality, and its own native user interface in Quattro Pro did not change the fact that it also achieved behavioral equivalence with the Lotus program and was a kind of clone.[364] Near-clones should be regulated for a market-preserving period of time just as clones would be.

U.S. 1031 (1987); *Borland,* 831 F. Supp. 223 (finding infringement from copying of 1-2-3 command structure); *Paperback,* 740 F. Supp. 37 (finding infringement because of copying virtually whole user interface of Lotus 1-2-3); *Softklone,* 659 F. Supp. 449 (finding infringement because of similarities in layout of commands for communications software). Some recent cases have suggested that similarities in behavior should not be considered a basis for copyright infringement. See, e.g., Computer Assocs. Int'l v. Altai, Inc., 775 F. Supp. 544, 559–60 (E.D.N.Y. 1991), aff'd in part, vacated in part on other grounds, 982 F.2d 693 (2d Cir. 1992). If the First Circuit rules that similarities in program behavior are not appropriate bases for finding infringement in the *Borland* case, it is possible that clones may begin to reappear in the software market. In our view, clones should be blocked from the market for a time, but it would undermine the norms of the competition policy underlying intellectual property law to block unpatented technical innovations for the 75 year term that copyright law provides to protect the expressive elements of copyrighted works.

359. See supra note 29 and accompanying text. Clones are generally easy to detect because of the identity of elements that the cloned product will have with the product being cloned. Some of the early clones were produced by firms that chose product names indicating the nature of their products. See, e.g., *Softklone,* 659 F. Supp. at 453 (clone of popular communications program was named "Mirror"). Mosaic Software named its clone of Lotus 1-2-3 "The Twin." See Edelman, supra note 29, at 30.

360. A near-clone is a program that attains functional equivalence or near equivalence with another program, even if there is some variation in the ordering of compilation elements.

361. See *Borland,* 831 F. Supp. 223.

362. See id. at 228–29. A file in the Borland program contained a representation of the Lotus command hierarchy in a scrambled order, with each command term designated by one letter.

363. See id. at 229–31.

364. Borland attempted to distinguish itself from Lotus Dev. Corp. v. Paperback Software Int'l, Inc., 740 F. Supp. 37 (D. Mass. 1990), by arguing that its was not a clone. See, e.g, Brief of Defendant/Appellant Borland Int'l Inc. at 4–16, Lotus Dev. Corp. v. Borland Int'l, Inc., No. 93-2214 (1st Cir. appeal docketed Nov. 19, 1993). On the other hand, Borland's product clearly reproduced the functionality of the Lotus product. See *Borland,* 831 F. Supp. at 229–31. It is noteworthy, however, that Borland's product came into the market in 1987, whereas Lotus 1-2-3 entered the market in 1983. Depending on the length of blockage that a market-oriented legal regime would provide, Borland might

The facts of *Whelan Associates v. Jaslow Dental Laboratory, Inc.*[365] present a good example of partial cloning (that is, cloning of subcompilations). Rand Jaslow copied the detailed procedure for how information should flow through Whelan's Dentalab program, as well as the detailed manner in which five subroutines performed their functions.[366] In doing so, he copied industrial design elements of Whelan's program, some of which were internal design elements (the information flow) and some external design elements (how the subroutines behaved). Jaslow attempted to achieve functional equivalence with these parts of Whelan's program without the skill or effort of independent development. We regard neither as copyright infringement. However, Jaslow's cloning of the information flow and partial cloning of the behavior of five subroutines were objectionable if they undercut Whelan's lead time for the incremental innovations she had introduced in her product.[367]

A market-oriented legal regime aimed at protecting industrial design elements of programs would not have an equivalent to copyright's "merger" doctrine[368] that would disqualify from protection highly efficient industrial designs embodied in programs or subcompilations that had become de facto standards in the marketplace.[369] We agree with Judge Keeton that functionally optimal program designs and aspects of programs that have become de facto standards may need some legal pro-

not have been blocked from using the Lotus command heirarchy because the blocking period might have expired. However, it might, for a time, have had to pay a reasonable royalty for this use of Lotus's innovation. As Section 7.5 will explain, we regard payment of reasonable royalties as an element of a market-oriented legal regime that should be available to innovators who register their innovative software designs.

365. 797 F.2d 1222 (3d Cir. 1986), cert. denied, 479 U.S. 1031 (1987). Recall that differently structured texts can produce the same behavior, see supra notes 22–24 and accompanying text, so there is no necessary relationship between the cloning of behavior and of information flow.

366. See id. at 1245–48; Whelan Assocs. v. Jaslow Dental Lab., Inc., 609 F. Supp. 1307, 1314 (W.D. Pa. 1985), aff'd, 797 F. 2d 122 (3d Cir. 1986), cert. denied, 479 U.S. 1031 (1987).

367. From a market-oriented standpoint, it would be significant that Whelan (with Jaslow's help) began marketing the Dentalab program written in EDL in 1979. Jaslow began its development effort in 1982 and began marketing its similar product in BASIC in 1983. See *Whelan*, 609 F. Supp. at 1315.

368. See supra note 197 and accompanying text for a discussion of the merger doctrine. Nor should "merger" or a kindred rule apply to software in the rare instance in which there is truly no other way to perform a function than the method used by the first comer. The short duration of protection available from a market-oriented regime would only give the first comer a lead time advantage for having been the first to discover the one way to do some task. However, independent discovery of the same one way would, at least for unregistered material, not be subject to liability.

369. See supra note 197 and accompanying text for a discussion of Professor Menell's suggestion that the merger doctrine should be employed to limit copyright protection for highly efficient software design elements and software components that had become de facto standards. While we believe Menell's proposal is economically more sensible than the overprotection that copyright would give to efficient or de facto standard elements of programs, his proposal logically leads to the underprotection of efficient software designs.

tection.[370] It is, after all, the goal of every programmer to design functionally optimal programs or program components that become de facto standards in the marketplace. However, traditional principles of copyright law, as well as competition policy more generally, call for withholding copyright protection from functionally optimal program designs and de facto standards.[371] This is why we argue for a market-oriented legal regime. Under such a regime, competitors' need to use efficient designs or emergent standards from other firms' programs would be satisfied by the short period of protection that a market-oriented regime would provide and by rights to use under standard license fees.

5.4.3 Of Substantially Similar and Substantially Different Program Compilations

While it is easy to decide to protect software developers against clones of their programs and not to protect against different programs that have been substantially independently created, the range of cases between these extremes presents more difficult choices.[372] Traditionally, copyright places its threshold low and protects against any work that is substantially similar and not independently developed.[373] Patent, on the other hand, places its standard high and requires that an infringer's product be substantially identical to the claimed invention on an element-by-element basis without regard to the dependency of the second comer's product.[374] A new regime for software innovations could choose one of these models or some compromise between them.

We favor a substantial identity standard for software. Copyright's substantial similarity standard may be appropriate for artistic or fanciful works where the creator's possibilities are nearly infinite and expression of an author's personality is highly valued. Programs, however, are industrial in character and, like other products of engineering processes, they often combine preexisting elements in incrementally new ways.[375] Moreover, many subcomponents of program compilations are "generic" to

370. See, e.g., Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 57 (D. Mass. 1990).

371. See supra notes 197, 241–242.

372. Lotus 1-2-3, for example, was in many ways substantially similar to VisiCalc, yet it included many new features and improved on VisiCalc's design. See, e.g., Bill Machrone, Roots: The Evolution of Innovation, PC Mag., May 26, 1987, at 166, 168–70.

373. See supra note 282.

374. Patents may be infringed if all elements of a patent claim are present in an accused device or, under the doctrine of equivalents, if there are equivalents to each of these elements in the accused device. See, e.g., Graver Tank & Mfg. Co. v. Linde Air Prods. Co., 339 U.S. 605, 607 (1950); Pennwalt Corp. v. Durand-Wayland, Inc., 833 F.2d 931, 935 (Fed. Cir. 1987), cert. denied, 485 U.S. 961 (1988); Martin J. Adelman & Gary L. Francione, The Doctrine of Equivalents in Patent Law: Questions that *Pennawalt* Did Not Answer, 137 U. Pa. L. Rev. 673, 700, 728–29 (1989); Harold C. Wegner, Equitable Equivalents: Weighing the Equities to Determine Patent Infringement in Biotechnology and Other Emerging Technologies, 18 Rutgers Computer & Tech. L.J. 1 (1992).

375. See supra notes 67–74 and accompanying text.

compilations of that kind.[376] A legal regime that protects this kind of artifact should be careful not to interfere with the kind of incremental reuse that is crucial to the practice of engineering.[377] Even copyright law has traditionally extended a relatively thin scope of protection, akin to the anti-cloning approach discussed here, to works whose content is predominantly functional.[378]

An anti-cloning standard would also be more predictable than a substantial similarity standard. Innovators, imitators, and courts can all be expected to recognize near-clones without much difficulty. Nonetheless, several variations are worth considering and the final standard should be open to discussion.

We note, for instance, that the Semiconductor Chip Protection Act (SCPA)[379] adopted a substantial similarity standard for judging infringement of chip designs,[380] although it did so with interesting twists that may also be applicable to program innovations. Under SCPA, if a second comer undertakes legitimate reverse engineering and decides, after reasoned consideration and substantial toil, that it should use a substantially similar design, there will be no infringement unless the two chips are substantially identical.[381] In other words, the SCPA gives engineers additional leeway where practical considerations are paramount.

The SCPA infringement standard also considers whether the second comer has introduced improvements, again varying the substantial similarity test to encourage cumulative innovation.[382] Whether a second

376. See Brown Bag Software v. Symantec Corp., 960 F.2d 1465, 1475–77 (9th Cir. 1992).

377. See supra notes 72–74 and accompanying text. See also Machrone, supra note 372, at 166–67 (explaining how software developers build on previous work of other developers).

378. See supra notes 182–189 and accompanying text; see also LaST Frontier Report, supra note 1, at 18–19 (discussing the application of copyright law to computer programs).

379. Pub. L. No. 98-620, 302, 98 Stat. 3347 (1984) (codified as amended 17 U.S.C. §§ 901–914).

380. SCPA, like the copyright statute, does not explicitly set out the standard, but the intent of Congress is evident from the legislative history. See H.R. Rep. No. 781, 98th Cong., 2d Sess. 1 (1984), reprinted in 1984 U.S.C.C.A.N. 5750. Courts interpreting the SCPA have applied it. See, e.g., Brooktree Corp. v. Advanced Micro Devices, Inc., 977 F.2d 1555, 1564 (Fed. Cir. 1992).

381. The reverse-engineering privilege appears in SCPA at 17 U.S.C. § 906 (1988). The intent to impose a substantial identity standard when a second comer has engaged in legitimate reverse engineering is evident in the legislative history of this provision. See Explanatory Memorandum, Mathias-Leahy Amendments to S. 1201, 130 Cong. Rec. S12, 916–17 (daily ed. Oct. 3, 1984). The Court of Appeals for the Federal Circuit relied on this history in reviewing an infringement decision applying the heightened standard. See *Brooktree*, 977 F.2d at 1566–67. For a further discussion of this approach to judging infringement, see Leo J. Raskind, Reverse Engineering, Unfair Competition, and Fair Use, 70 Minn. L. Rev. 385, 406 (1985).

382. See Raskind, supra note 381, at 398–402 (citing legislative history in support of this idea). See also *Brooktree*, 977 F.2d at 1569 (affirming finding of SCPA infringement in part because of insignificant differences in defendant's chip design).

comer who improves while imitating should be less subject to sanction than one who simply imitates is a subject worthy of further discussion. Under an anti-cloning approach, we think improvements are unlikely to matter. But if program compilations are subject to a substantial similarity rule, a market-oriented legal regime might give some weight, as SCPA does, to whether the second comer introduced improvements.

One final issue is whether to treat the migration of innovations to remote markets or applications differently than adjacent use. Many software innovations have broad ranges of application. Features, algorithms, or subcompilations of features may migrate to domains entirely different from those in which they originate. In the process, imitators may transform the original innovations significantly in order to integrate them into new contexts.

Whether a legal regime should block such migration at all, block it for a shorter time than adjacent imitation, or only require royalty payment is an important question. It is desirable to ensure that even remote market participants do not reap the benefits of another's innovation without contributing towards the cost of its initial development. Yet, transplanting innovations to a wholly different market often requires additional creativity and has less potential to affect the innovator's own market. Accordingly, one might decide that reuse in new applications or remote markets should be permitted sooner or with shorter compensation periods than should reuse in applications and markets close to those in which an innovation was originally introduced.

5.4.4 Interoperating Programs

In order for a program to interoperate with another program, it must reproduce some or all of the other program's compilation of interface information. Whether interfaces can or should be protected by existing legal regimes has been the subject of much controversy in the United States and abroad.[383]

While we have no quarrel with the U.S. copyright decisions that have permitted reuse of interface information,[384] courts in these cases have ignored the fact that internal interfaces are compilations of information

383. See McManis, supra note 329, at 45–50 (discussing intellectual property protection and reverse engineering in both the U.S. and E.C.); Pamela Samuelson, Comparing U.S. and E.C. Copyright Protection for Computer Programs: Are They More Different Than They Seem?, 13 J. L. & Comm. 279, 285–92 (1994) (discussing protection of program interfaces in the aftermath of Sega Enters. v. Accolade Inc., 977 F.2d 1510 (9th Cir. 1992) and Computer Associates Int'l v. Altai, Inc., 982 F.2d 693 (2d Cir. 1992) and comparing with E.C. law).

384. See, e.g., *Altai*, 982 F.2d at 693 (holding elements of programs constrained by external factors such as the hardware or software with which it will interoperate are unprotected by copyright); *Sega*, 977 F.2d at 1510 (ruling that functional requirements for achieving compatibility are unprotectable by copyright).

that require considerable creativity to produce.[385] Not only are interfaces valuable, they are also often very costly to develop and embody considerable innovation. We regard internal interface compilations as unprotectable by copyright law because they are industrial compilations of applied know-how, information equivalents to the gears that allow physical machines to interoperate.[386] It is a separate question whether this sort of industrial compilation should be subject to a blocking period or right of compensation such as we recommend for behavior and other program compilations.

There are three principal ways that firms presently achieve interoperability. First, some developers publish interface specifications to enable third party developers to construct programs that will interoperate with their programs.[387] Firms have incentives to do this when they think they can increase sales of their software or hardware systems, which will be more in demand if there are many programs that interoperate with their system.

---

385. The argument in favor of copyright protection for internal interfaces emphasizes that selecting and arranging program interface elements requires creativity that easily meets the minimal creativity standard set forth in Feist Publications v. Rural Tel. Serv. Co., 499 U.S. 340, 345 (1991). As long as a programmer's initial choices were not wholly dictated by function, proponents of copyright protection for interfaces argue, the compilation is original and should be protected against copying, whether of a literal or nonliteral nature. See, e.g., Lake et al., supra note 335.

The initial design of an interface will generally not be significantly constrained unless standards have been promulgated. Where standards have been promulgated, there may be relatively little opportunity for creativity in expression. See, e.g., Secure Servs. Technology, Inc. v. Time & Space Processing, Inc., 722 F. Supp. 1354 (E.D. Va. 1989) (describing protocol governing "form, timing, [and] order" of digital signals and thereby limiting permissible variations). On the other hand, once an innovator has developed an interface, the design of that interface will serve as a constraint on any developer that wants to develop an interoperating program. See, e.g., *Altai,* 982 F.2d at 698–700, 714–15 (both plaintiff's and defendant's programs constrained by need to interact with IBM operating systems programs). Proponents of copyright protection for interfaces find support in Apple Computer, Inc. v. Franklin Computer Corp., 714 F.2d 1240 (3d Cir. 1983), cert. denied, 464 U.S. 1033 (1984), wherein the court indicated that the desire to achieve compatibility "is a commercial and competitive objective which does not enter into the somewhat metaphysical issue of whether particular ideas and expressions have merged." Id. at 1253.

386. See supra notes 35–36 and accompanying text.

387. Perhaps the best-known early example of this was the decision by IBM to make public the specifications for its IBM PC. See Ferguson & Morris, supra note 80, at 52.

> The IBM PC was also the first deliberately 'open' computer architecture, a fundamental insight that shaped the future of personal computing. From the very start, Boca Raton [where IBM developed the PC] recognized that the best way to make the PC the industry standard was to publish all its technical specifications and make it easy for third parties to build add-on devices or write PC software applications, a principle that took Apple years to understand.

Id. at 29. Of course, the publication of this information also enabled other computer manufacturers to make IBM-compatible computers.

Second, even when firms do not publish their interfaces, they often make the information available on a licensing basis.[388] A firm's willingness to license its interface either with or without royalties will generally turn on its expectation that consumers will value its products based, in part, upon the number of programs that interoperate with them, although royalties and license fees can be significant sources of revenue. Firms that want fast and low effort access to interoperability information have incentives to license it.[389]

Third, firms can decompile another firm's program, extract the interface information, and then reimplement it in different code.[390] Decompilation is the only route for obtaining this information when other firms are unwilling to license their interfaces.[391] But some firms that could license interface information choose not to do so, having decided that they would rather pay the costs of reverse engineering than a license fee.[392]

Decompilation and reimplementation of an interface are sufficiently arduous processes that they necessarily entail significant delay, leaving the firm whose interface information is being appropriated a nontrivial head-start in the marketplace.[393] As a consequence, we do not perceive

388. IBM and Fujitsu historically had a licensing arrangement by which IBM would provide interface information to Fujitsu, subject to payment of royalties. When Fujitsu decided to seek the same information by decompilation, IBM challenged the action. The resulting arbitration produced a licensing arrangement under which IBM would provide interface information to Fujitsu subject to a royalty-bearing licensing. See International Business Machines Corp. v. Fujitsu Ltd., American Arbitration Association, Commercial Arbitration Tribunal, Case No. 13-T-117-0636-85 (Nov. 29, 1988) (Mnookin & Jones, Arbs.).

389. Sega and Nintendo both have large numbers of licensees who pay for the privilege of producing games that will operate in their videogame consoles.

390. See, e.g., Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1514–15 (9th Cir. 1992).

391. Licenses may be unavailable to firms seeking the interface information in order to develop a competitive substitute for the program containing this information, rather than just a program that will interoperate with it.

392. Atari Games, for example, had been a Nintendo licensee, but eventually decided it wished to continue to produce Nintendo-compatible programs without paying license fees. It consequently decompiled the Nintendo program to learn how to make a compatible program without Nintendo's aid. See Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832, 836 (D.C. Cir. 1992).

393. On remand, the District Court in *Atari* offered a market-oriented rationale for distinguishing between the lawfulness of achieving compatibility with present versions of Nintendo games and the unlawfulness of achieving future compatibility. Had Atari limited itself to achieving compatibility with present versions of Nintendo's system, Nintendo would have had lead time protection for each new release because non-licensees would have to decompile each new version to reachieve compatibility. The court thought that allowing a non-licensee to make a program that would achieve compatibility with future versions would undermine Nintendo's incentives to produce innovative hardware consoles. See Atari Games Corp. v. Nintendo of America Inc., 30 U.S.P.Q.2d (BNA) 1401, 1407–08 (N.D. Cal. 1993). While we agree that the appropriation of interface information should be judged by market-oriented standards, courts deciding copyright cases should not focus on whether manufacturers of hardware consoles have enough leadtime to induce

that market failure is presently occurring when software developers appropriate interface information.

However, if decompilation technology improves substantially, there may be a need to consider whether use of another firm's internal interface information should be blocked for a market-preserving period of time, or perhaps be subjected to a right of compensation.[394] Internal interfaces are, after all, industrial compilations requiring skilled effort to create. A second comer who wants rapid access to them should have to contribute toward their development expenses.

5.4.5 Add-On Software

"Add-on" software typically offers some enhanced functions to an existing program, either modifying the other program's behavior or supplementing it in some way. Galoob's Game Genie, for example, permitted users to change certain aspects of the play in Nintendo videogames.[395] A program originally named "Sideways" permitted users of Lotus 1-2-3 to print spreadsheet data sideways on standard-sized paper when they preferred that format to the conventional one Lotus provided.[396]

To modify or supplement an existing program's behavior, an add-on program must be able to interact with that program. To that extent, add-on software is a kind of interoperating program, although the extent of interoperability needed is typically less than that required for an operating system program and its application programs. The Sideways program, for example, had to be able to "read" file formatting information used by the spreadsheet program, but did not need to interact with other parts of the program.

---

investments in hardware innovation. See Richard H. Stern, Reverse Engineering for Future Compatibility, 16 Eur. Intell. Prop. Rev. 175, 178–80 (1994).

394. Perhaps a market-oriented legal regime should distinguish between those who seek only to interoperate with and those who want to develop a competitive substitute for the target program. From a market standpoint, the competitive substitute program has a greater potential to undermine incentives to innovate than would a complementary interoperating program. In a market-oriented regime, the developer of a complementary product might, for example, be permitted to enter the market sooner or more cheaply than a direct competitor because it and the program with which it interoperates compete in different markets.

During the debate about the European directive on legal protection of computer programs, there was considerable discussion about whether the directive should permit decompilation only to develop a program that would interoperate with the decompiled program. Such a policy would have meant that a firm that decompiled another firm's program in order to make a competing substitute would infringe copyright. In the end, EC policymakers decided not to make such a distinction, although it requires a somewhat convoluted understanding of the Directive's provisions to discern this. See Vinje, supra note 328, at 73–76.

395. See Lewis Galoob Toys, Inc. v. Nintendo of America, Inc., 964 F.2d 965 (9th Cir. 1992), cert. denied, 113 S. Ct. 1582 (1993).

396. Lotus eventually acquired rights to this program and integrated its functionality into 1-2-3.

An exclusive property rights regime might take a broad view of derivative work rights, regarding the add-ons as a market that should be reserved to the firm whose product they modify or complement.[397] Add-ons are, after all, essentially parasitic in nature. They rely heavily on the value of the program whose functionality they modify or enhance. Almost by definition, there is no market for add-on software as a stand-alone product. Furthermore, add-on programs usually reproduce some elements of the underlying program, such as portions of its interface or file structures, in order to interoperate with it.

In the software industry, add-on software is very common and is widely regarded as market- and competition-enhancing.[398] From a market-oriented perspective, the add-on program's developer, no less than the developer of the underlying program, must have a market incentive for investing in cumulative innovation.[399] The Ninth Circuit took this view in *Galoob*, noting that anyone who purchased Galoob's Game Genie would already have purchased videogames from Nintendo.[400] If the Game Genie enhanced a user's pleasure in Nintendo games, the court saw no reason to interfere with this enjoyment by blocking a complementary product from the market. Yet, one can argue, from a market-oriented standpoint, that the developer of an add-on program builds on the underlying developer's research and development and should not be entirely exempt from contributions to these costs.

6 Goals and Principles for a Market-Oriented Approach to the Protection of Innovative Technological Know-How

Having set forth the three primary factors by which to determine whether an appropriation of program know-how arising from behavior would induce market failure, we know when to act. We turn now to discuss the goals and principles for a legal regime that could implement this anti-cloning approach in a technologically appropriate and market-oriented manner. We conclude with a set of abstract elements that we think

397. Nintendo made such an argument in *Galoob*, 964 F.2d at 967–69. Even Judge Keeton, who has otherwise taken a very expansive view of the scope of copyright protection in programs, views add-on software as lawful. See Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 78–79 (D. Mass. 1990). Had Paperback merely sold a database product that interacted with the Lotus program, the judge would have considered it a lawful, incremental addition to the field. However, by making a directly competing program that had a database capability as well, Paperback had crossed the line of illegality.

398. Apart from *Galoob*, 964 F.2d at 965, there have been no other lawsuits challenging the legality of software add-ons.

399. See Christian H. Nadan, Comment, A Proposal to Recognize Component Works: How a Teddy Bears on the Competing Ends of Copyright Law, 78 Cal. L. Rev. 1633 (1990) (taking market-oriented perspective on legality of add-on programs, which author describes as component works).

400. See *Galoob*, 964 F.2d at 971; see also New York Times Co. v. Roxbury Data Interface, Inc., 434 F. Supp. 217 (D.N.J. 1977) (copying information to develop an index to newspaper's indices was fair use, in part because it did not supplant the market for them).

any new regime should include. Sections 7 and 8 will use these principles as a basis for assessing the best option among possible legal mechanisms for protecting program know-how. We recognize that these principles cannot all be fully achieved in any one legal regime. Our goal was to achieve a balance among competing or conflicting principles and to identify a regime that would satisfy as many principles as possible.

6.1 Goals and Principles of a New Regime

6.1.1 Do not try to wipe the slate clean; build on existing legal foundations.

The idea of constructing one legal regime that could protect all aspects of software might be attractive from a theoretical standpoint. However, given the disruption it would cause to settled expectations, a totally new approach to protection of all aspects of software may be infeasible. Such a total revision may also be unnecessary because in some respects, existing legal regimes appropriately protect software innovations without distortion of their basic principles. Copyright law, for example, has provided a simple and effective means of deterring wholesale copying of source and object code and of expressive texts, pictures, or audiovisual material generated when program instructions are executed.[401] Any new legal regime should supplement protection available from existing legal regimes without overlapping with them.

6.1.2 Focus on solving the most serious problems.

No legal regime can solve all problems that industries may have. Nor can a legal regime solve them perfectly. Our goal was to concentrate on a workable solution to the most serious problems confronting the software industry, and to avoid getting distracted by the difficult questions presented by marginal cases.

6.1.3 Provide a well-lit playing field.

A legal regime that protects program behavior and industrial design elements responsible for it should be reasonably predictable as to scope and duration of protection. This will encourage investment and reduce the potential for litigation.

6.1.4 The regime should flow from and be responsive to the nature of the technology protected.

An appropriate legal regime for protecting the know-how embodied in computer programs should be geared to protection of the true sources of value in software: behavior, the industrial designs that produce behavior, and conceptual metaphors.

401. See supra notes 166–168.

6.1.5 The regime should make legal distinctions that are technically coherent.

If a legal regime for the protection of software innovations does not make technically coherent legal distinctions, questions phrased in a legal context may be impossible to answer meaningfully when presented to technical witnesses or experts.

6.1.6 The regime should be, so far as is possible, independent of the current state of technology.

The legal regime should not be based upon or otherwise depend in substantial ways on the current state of technology, but should be capable of evolving naturally as the technology itself evolves.

6.1.7 The regime should also encourage disclosure and dissemination of program know-how, facilitating improvements, and new applications.

A legal regime for protecting applied know-how in programs from market-destructive appropriations should not block access to that know-how, but merely regulate its use. There should be room to improve existing applications and to transfer know-how to new application domains.

6.1.8 The regime should encourage product-enhancing innovations and discourage overheated innovation.

A product-enhancing innovation is one that either improves the product's utility without unduly increasing its costs, or reduces its costs without impairing its utility, or both. Overheated innovation is variation on a product that does not materially add to its utility or advance the level of skill in the market, but instead is motivated by the desire to appear different in order to fit into or avoid legal categories.[402] True innovation suffers when manufacturers fear to invest in research and development likely to lead to improvements or cost reductions and invest instead in marginal differentiation.

6.1.9 A market-oriented legal regime should encourage innovation by avoiding market failures.

We believe that innovation will arise naturally, given an opportunity to flourish. The role of the marketplace is to provide a forum for natural selection in the ongoing breeding of innovation. The primary function of a market-oriented legal regime would be to prevent market failure that

402. See, e.g., Richard S. Marcovits, The Allocative Efficiency of Encouraging Additional Real Investment in General and R&D in Particular: A Preliminary Third-Best-Allocative-Efficiency Analysis of Various Proposed Policies (1994) (unpublished manuscript, on file with the Columbia Law Review) (discussing policies affecting research and development that produce inefficient investments in product variety).

discourages investment in both long-term and short-term software innovation.

6.1.10 A market-oriented legal regime should prevent market failure by providing innovators with reasonable lead time.

Products imitating an innovation should not be able to arrive faster in the marketplace than necessary to provide innovators with reasonable incentives to invest in new or improved products. The developer of an innovative software product may not ultimately be successful in the marketplace, but should at least have a period of unobstructed opportunity to seek market reward before imitations can lawfully appear there.

Artificial lead time of this sort would be considerably shorter than the duration of exclusive rights granted by patent and copyright law and more in keeping with the lead time that classical trade secrecy law has provided.

6.1.11 To provide an appropriate scope and duration of protection, a market-oriented 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.[403] For software, it is currently the one or two years required to create and to test a totally new product, or the roughly twelve to eighteen month interval required to develop and to test the new release of a preexisting product. The market rate is an empirical phenomenon dependent on a wide range of factors, including the underlying technology (e.g., how long it takes to develop and debug a program) and human psychology (e.g., how soon consumers will buy an upgrade after purchasing the original program). The rate can change depending on the state of innovation in the market, the pace of that innovation, and the relative maturity of the market.[404] The market rate should also be considered in setting an appropriate duration for protection of program know-how.

A market-oriented regime would allow members of the relevant technical community operating within public interest guidelines to adjust the legal regime so that it is attuned to the rate of development in the market. This flexibility can help avert cycles of overheated and chilled innovation.

403. The phrase is ours. For a discussion of the idea, however, see William Kingston, The Unexploited Potential of Patents *in* Kingston, Direct Protection, supra note 67, at 286.

404. Consumer pressure can contribute to the slowing of the new release cycle. The Massachusetts Software Council reports the following discussion by a software company CEO: "At the beginning of our applications business, we also committed to a release every six months. It was one of the biggest things we promoted and sold. Finally it came to a crisis point because the product was becoming so unstable. . . . I . . . stood up in front of our customers . . . and said . . . 'we're not going to do releases every six months any more.' [I got a] Standing ovation." Business Practices, supra note 265, at 19.

6.1.12 A market-oriented legal regime should provide an opportunity for innovators to recoup their research and development expenses and earn a return on their investment insofar as the work results in valuable innovation.

A market-oriented legal regime for protecting software innovation would provide innovators with an opportunity to recoup their research and development expenses and earn a return on investment. It would not, however, reward effort alone, or expenditure alone, because this has too much potential to encourage inefficiency. The focus should instead be on stimulating investment in productive innovation by ensuring the innovator an opportunity for a headstart in the marketplace. One factor that should be considered in determining a reasonable duration of protection for the know-how arising from program behavior is the industry's assessment of the duration that would permit recoupment of efficient expenditures on research and development.[405]

6.1.13 A market-oriented legal regime should avoid wasteful reduplications of effort.

Substantial societal costs are incurred when program know-how is kept as a trade secret. Some of these arise from the costs of maintaining secrecy; others derive from the expenditures directed at reverse engineering or engaging in other efforts to duplicate or independently recreate the know-how. One mechanism that allows these costs to be spread is licensing. Insofar as software products bear their know-how on their face, there is a problem: No one wants to pay for what they can get for free by looking at the product as it is offered in the marketplace.[406] Hence, current legal regimes do not provide a framework for software-technology licensing agreements that permit underlying research and development costs to be spread.

6.1.14 Market participants should share research and development costs in a market- and competition-enhancing way.

It may not be efficient for every participant, or would-be participant, in the market to make substantial investments in research and development work. But it may be both efficient and market-preserving to require contributions to the original research and development firms that want

405. Fixing a duration for the legal protection of program innovations will inevitably only approximate a period of recoupment that the software industry would find reasonable.

406. See Cooter & Ulen, supra note 14, at 112–13. There is at least one social benefit to compiled know-how borne on the face of a product: the socially wasteful expenditures that attend trade secrecy are avoided. Eliminating these wasted costs is market-enhancing because the market doesn't have to pay for them again and again. But if the vulnerability of applied know-how on the face of a product impairs the ability of its developer to recoup costs of development and make sufficient profits to permit further innovations to occur, it can be market-destructive as well.

to use the fruits of others' intellectual labor to make products that embody follow-on incremental innovation.

One way to accomplish this goal would be to put firms not engaging in substantial research and development of their own to the business choice of either contributing to the costs of any research and development results they want to appropriate, or refraining from appropriating those results for a market-preserving blocking period. An appropriate period would be short enough that entrepreneurs feel they are faced with a plausible decision about whether to license the innovation or wait for its emergence into the public domain.

6.1.15 A market-oriented legal regime should recognize that an innovation may be distinct from a specific product embodying it and provide a means for innovators unsuccessful in their own commercialization to obtain some remuneration for their contributions to the market.

The market may recognize the value of an innovation even if it does not value the specific product in which the innovation is first embodied.[407] We wish to encourage market-enhancing innovation in any product and in any market. To do so requires an explicit recognition that an innovation embodied in a product, and the skilled know-how required to make it, may be distinct from the product itself. A market-oriented legal regime should be willing to uncouple the two and treat innovations in applied know-how as valuable.

Many commercially important incremental innovations in software first appeared in products that were themselves commercially unsuccessful. Sometimes the market was not ripe for the first product embodying the innovation; sometimes the innovation achieved success only when combined with other components; and other times the innovator was simply unskilled in making the innovation into a product or in marketing it. A market-oriented legal regime should find a way to spur valuable innovation, independent of whether its developer embodies it in a successful product.

407. Examples abound. Features of the graphical user interface introduced in Xerox PARC's Alto product achieved success in the marketplace when reimplemented by Apple Computer, although Alto itself was not a successful product. See supra note 86 and accompanying text. Lotus's Symphony program introduced to the market in February 1984, incorporated a spreadsheet, a graphics feature, a word processor, a database manager, and a telecommunications feature into a single program. See Manes & Andrews, supra note 259, at 248. Although this product failed, later "suites" of combined programs have succeeded in the market, including Microsoft's Office for Windows, introduced in October 1990. In 1984, IBM had internally developed a graphical user interface called Mermaid that, combined with its CP/X86 operating system, offered a considerable amount of the functionality that would later appear in Microsoft's Windows 3.0 in 1990, including multiple windows, multitasking, and virtual memory. It scrapped the program, however, in the course of the negotiations with Microsoft that led to the development of OS/2. See Ferguson & Morris, supra note 80, at 72–74.

6.1.16 A market-oriented legal regime should provide incentives to agree rather than to litigate.

A market-oriented legal regime should create incentives for an innovator and someone who wants to use the innovation to reach agreement, rather than to litigate. Licensing among software developers is already quite common. Developers routinely make agreements with royalty provisions when they want to package another firm's product along with theirs or to incorporate applied know-how from another firm's software product into theirs.[408] A market-oriented legal regime could more directly encourage this practice, for example, by developing standard licensing arrangements that market participants would be free to contract around.[409]

6.1.17 A market-oriented approach should be multi-faceted enough to distinguish among different kinds of second comers.

A market-oriented legal regime might consider a number of factors in determining whether second comers should have to pay a standard fee to use an innovation or should be blocked from use for a reasonable period of time. Such factors might include the relative size of the appropriated innovation, the manner by which the taker obtained access to it, the degree of similarity between the works, the extent to which the second product improved on the first, and the proximity between markets in which the innovator and second comer were operating.[410] A multi-faceted approach would distinguish among different kinds of second comers, according to the market effects of their borrowings.

6.1.18 A market-oriented legal regime should, as far as practicable, minimize the costs of obtaining protection.

We use the term "self-executing" to refer to a market-oriented legal regime that would provide protection with a minimum of bureaucracy, time, or overhead expense. The more self-executing a legal regime is, the more "market-friendly" it is likely to be. One way to achieve both market protection and self-execution is to provide some degree of lead-time protection automatically. Any formal requirements should be tailored to what is reasonable in market terms. This is especially important in rapidly-developing technology fields, such as software.[411]

408. Microsoft Word 5.1 for Macintosh, for example, bundles grammar- and spell-checking programs developed by independent companies in its word processing package. Their separate copyrights are acknowledged in the same opening screen that gives notice of Microsoft's copyright in Word. Houghton-Mifflin is credited as the copyright owner for the International CorrectSpell, Houghton-Mifflin Co. and Language Systems Inc. as copyright owners of CorrectText, and Soft-Art, Inc. as copyright owner of the thesaurus program.

409. See supra note 250 concerning the high transaction costs of licensing,

410. See supra Section 5 for discussion of these dimensions.

411. See supra note 134.

6.1.19 A market-oriented legal regime should minimize barriers to entry.

In curing market failure attributable to the rapid appropriability of know-how in software, care should be taken not to induce another kind of market failure by artificially raising barriers to entry. Innovation in the software market will, in general, be enhanced by increasing opportunities to participate in it.

6.1.20 A market-oriented legal regime should promote consumer welfare.

The goal of avoiding market failure in the software industry is not an end in itself, but rather a means for promoting consumer welfare more generally. The benefit from efficient incentives to innovate will accrue not only to software developers, but also to the public, because a larger number of desirable products will be available in the market. This goal also means that such a legal regime should be as wary of overprotecting program innovations as it is about the underprotection problem that has been the main focus of this Article. Indeed, the greatest threat to the adoption of a market-oriented legal regime may come from those who have actively strived in recent years to stretch existing legal regimes to protect software innovations in order to avoid underprotecting software. They may be reluctant to accept a substantially lesser duration of protection than existing regimes provide, even if this is all that is necessary to repair market failure and promote consumer welfare. By emphasizing consumer welfare as a principle and goal of a market-oriented legal regime, however, it should be possible to overcome the temptation to overprotect.

6.2 Abstract Elements of a New Regime

Emerging from our discussion of the principles discussed in this Section, we reached consensus that a market-oriented framework for protecting program innovations should be built around a number of abstract elements that Professor Reichman articulates as follows:[412]

1. Treat industrial compilations of applied know-how as the objects of protection
2. Provide artificial lead time to overcome market failure
3. Develop a menu of user liabilities that sensibly allocate the costs of research and development among innovators and borrowers that constitute the relevant technical community
4. Allow registration with some disclosure
5. Supply pro-competitive standard ground rules
6. Develop legal and organizational means to enable collective action to enforce and adjust the liability framework

412. See Reichman, Legal Hybrids, supra note 100, at 2545.

Reichman views these elements as useful components in a more general approach to constructing a legal regime for dealing with the appropriability of unpatentable and uncopyrightable know-how, regardless of the medium in which such know-how is embodied.[413]

7 Prototype Frameworks for a Market-Oriented Legal Regime to Protect Software Innovations

This Section will describe a number of prototype frameworks for implementing a market-oriented form of legal protection for software innovations consistent with the goals and design principles discussed in Section 6. Although no prototype was flawless, the approach described in Section 7.5 best approximated the abstract elements for a model legal regime and best satisfied our design principles. It would provide a short period of blockage against cloning, and a system by which developers of incrementally innovative software designs, such as conceptual metaphors or algorithms, could register their innovations and make them available for licensing.

We offer our views about each alternative we considered, on the theory that some of the "negative know-how" compiled during our search for an optimal legal regime could usefully contribute to a further discussion of the issues.

7.1 To Each According to Its Needs

Ideally, a market-oriented legal regime would protect each software innovation against commercial imitations just long enough to enable its developer to enjoy the same lead time as other innovators who contributed equal value to the market. The necessary amount of artificial lead time would depend on the amount of natural lead time already available, given the ease or difficulty of reverse engineering. If such individualized tailoring were possible, each innovator could count on the chance to earn exactly the return necessary to justify its investment. Despite the theoretical appeal of such an approach, we regard it as too unpredictable to be workable.

7.2 Automatic Blockage of Cloning

A more plausible approach would be to protect program behavior and other industrial design elements of programs against cloning for a period of time sufficient to avoid market failure. Protection against cloning might commence, by operation of law, from the first public marketing of the program embodying it.

This approach would satisfy a number of design principles. First, it would provide protection for software innovations that existing legal regimes cannot provide without the cycles of under- and overprotection.

413. See id. at 2544–45.

Second, it is a low-cost, self-executing approach. Third, because it would be limited to protecting against identical or near-identical copying, it would be relatively predictable.[414] Fourth, it would directly protect against the next most trivial means of acquiring functional equivalence after outright duplication of code, namely, identical or substantially identical copying of program compilations and the engineering designs that give them coherence. Fifth, because the duration of protection would be geared to preserving lead time, it would allow compiled know-how to be reused thereafter, promoting cumulative innovation, competitive add-ons, and the standardization of efficient solutions.

This approach is, however, not without drawbacks. For some, automatic protection only against cloning will seem too "thin" because substantially similar implementations would be unlikely to be deemed clones. This may be troublesome not only when the two products are competing, but also when a subcompilation migrates to a domain remote from its original context. Migration may require enough changes in a compilation's composition that the migrated subcompilation may not be deemed a partial clone even if it varied from the original only in ways dictated by the changed context. In addition, because small software developers tend to take longer to develop their markets than bigger companies, a short period of protection may better serve the needs of large companies than small ones. Also, without a registration system, it may be difficult for second comers to know when the anti-cloning period expires. Lastly, an automatic anti-cloning system will not get any compensation to the innovator whose own commercialization effort is a failure, but who has benefited the market by introducing an innovation that others commercially exploit with success.

7.3 Automatic Anti-Cloning Protection Followed by an Automatic Royalty-Bearing License

Reflections on these drawbacks led us to consider a two-phase protection regime that might satisfy more of the market-oriented design principles than an anti-cloning approach alone would do. The first phase would block clones in order to give innovators the opportunity to develop a niche in the marketplace. In the second period, others could use the innovation if they paid standard licensing compensation to the innovator. The second stage would ensure that some compensation would get to those who introduced an innovation to the marketplace when others sought to use it, regardless of whether the innovators' own implementation was a commercial success. The duration of such a second period should also be short under the market-based principles discussed above.

Although automatic licenses have not been much used in the United States, they are a common feature of industrial property laws in other

414. See supra Section 5.4.3.

countries.[415] Because of longstanding public policies favoring the reuse of publicly accessible technical information and incremental innovation,[416] it is worth keeping an open mind about an automatic license approach as a partial solution to the compiled know-how problem addressed in this Article.

Notwithstanding the theoretical appeal of this approach, practical problems would unquestionably impede successful implementation. Without a registration system to identify and delineate the subject matter to be protected, it would be difficult to know when blockage periods ended and when the automatic license period commenced. Transaction costs for implementing the licenses would likely be high unless the law encouraged or required licenses with standardized terms. There would also need to be some reliable way to establish appropriate royalties for different classes of use.

7.4 SCPA-Like, Automatic Anti-Cloning Protection Plus Registration Approach

Semiconductor chips, like computer programs, are an information technology that fits uneasily into the framework of traditional intellectual property law.[417] Not surprisingly, then, SCPA reflects many of the design principles we seek to implement for software. SCPA provides automatic anti-cloning protection to semiconductor designs from the date of the

415. For examples, see Reichman, Legal Hybrids, supra note 100, at 2465, 2477–78 (discussing British unregistered design right and Italian engineering law).

416. See supra notes 233–239 and accompanying text.

417. When it passed SCPA, Congress decided that semiconductor designs and the masks used as stencils to create chips were too functional to be protectable by copyright law. It recognized that patents could not give sufficient protection to semiconductor designs because chip design is largely an engineering enterprise, requiring skilled efforts to make incremental improvements in selection and arrangement of functional elements, rather than invention. See Copyright Protection for Imprinted Design Patterns on Semiconductor Chips: Hearings on H.R. 1007 Before the Subcomm. on Courts, Civil Liberties and the Admin. of Justice of the House Comm. on the Judiciary, 96th Cong., 1st Sess. 55–62 (1979) (statement of James M. Early, Division Vice President, Fairchild Camera and Instrument Corp.). As with software, semiconductor chip designs tend to bear much of the incremental technical innovation they embody on the face of the product sold in the marketplace. See Copyright Protection for Semiconductor Chips: Hearings on H.R. 1028 Before the Subcomm. on Courts, Civil Liberties, and the Admin. of Justice of the House Comm. on the Judiciary, 98th Cong., 1st Sess. 5–6 (1983) (statement of Rep. Norman Y. Mineta). As with software, chips are very costly to develop, but once developed, their designs are vulnerable to fast and inexpensive appropriations that undercut innovator's incentives to invest in innovation. See House Comm. on the Judiciary, Semiconductor Chip Protection Act of 1984, Report to Accompany H.R. 5525, H.R. Rep. No. 781, 98th Cong., 2d Sess. 2–3 (1984), reprinted in 1984 U.S.C.C.A.N. 5750, 5751–52. It requires, in other words, only relatively trivial effort for a second comer to acquire functional equivalence to an innovative chip design by copying products sold in the marketplace. See id.; see generally Samuelson, Lessons of Chip Law, supra note 5, at 491 (citing the disproportion in expense of development and ease of copying as main reason Congress passed sui generis protection for semiconductors).

first commercial distribution of a chip embodying them.[418] This protection lasts for two years unless a chip developer registers the design with the Copyright Office.[419] The SCPA registration process, like that for copyrights, involves only a light examination of the application before a registration certificate issues.[420] A timely registration will extend the duration of protection to ten years.[421] An additional inducement to registration is that the SCPA certificate, like that for copyright registrations,[422] constitutes prima facie evidence that the holder has SCPA rights.[423] Under SCPA, others are free to use aspects of a chip compilation as long as they do substantial independent design work in preparing their competing chips.[424]

Although it is fair to say that SCPA protects semiconductor designs, it does so indirectly. SCPA's subject matter is actually "mask works," that is, the set of stencils or "masks" used in the manufacture of chip layers under the technology in common use when SCPA was devised.[425] A set of "mask works" for a particular semiconductor chip design must accompany the application for registration sent to the Copyright Office.[426] A registration system has worked reasonably well for SCPA because mask works are an intermediate work-product of the manufacturing process that can accompany the registration application.[427]

418. Protection for a mask work commences on either the date it is registered with the Copyright Office, or the date on which it is first "commercially exploited" anywhere in the world, whichever is first. See 17 U.S.C. § 904(a) (1988).

419. See id. § 908(a).

420. See id. § 908(e); cf. 17 U.S.C. § 410(a) (1988) (copyright). The copyright registration approach postpones most of the costs of examination until infringement litigation, if any occurs. Patent issuance, by contrast, may take as long as three years and cost thousands of dollars. The primary interest protected by the SCPA are economic and material: promotion and creation of new technology, investment in technology, and business certainty. Congress believed these interests were best advanced by a fast, inexpensive system for establishing ownership rights. See Richard H. Stern, Determining Liability for Infringement of Mask Work Rights under the Semiconductor Chip Protection Act, 70 Minn. L. Rev. 271, 380 (1985).

421. See 17 U.S.C. §§ 904(b), 908(a) (1988).

422. See id. § 410(c).

423. See id. § 908(f).

424. See id. § 908(e); Raskind, supra note 381, at 395–98 (citing legislative history establishing requirement that copyist expend substantial energy and resources on reverse engineering and reimplementation).

425. See 17 U.S.C. § 902; see also id. § 901(a)(2) (defining "mask work").

426. See id. § 908(c); 37 C.F.R. § 211.5 (1993). The choice of mask works as the subject matter for the SCPA protection regime has been criticized because, as chip technology evolves, masks are less frequently used, making SCPA potentially obsolete. See Morton D. Goldberg, Semiconductor Chip Protection as a Case Study, *in* Global Dimensions, supra note 6, at 329, 332–33. It would probably have been better to conceive of the subject matter of SCPA as semiconductor chip designs and to make masks a form of identifying material regarding the design for registration purposes.

427. Registration of chip designs also remedies some defects of a pure anti-cloning approach. For instance, applications for registration must state the date on which commercial exploitation commenced. This helps copyists determine when legal

Registration of software innovations would not be as easy to achieve because there is no intermediate design document uniformly prepared by software developers for either internal or external designs that could serve as registration material. Developers of innovative programs would be reluctant to register a design document that disclosed all of the internal design elements of their programs, information that they can now protect as trade secrets because of the difficulties of gaining access to it by decompilation.[428] So, although SCPA has some features that might usefully be adopted in a legal regime for protecting software innovations, its registration system could be unworkable as applied to software.

7.5 Modified SCPA Approach: Some Automatic Protection Complemented by Registration of Innovative Elements

A modified SCPA approach, tailored to market-preservation principles, might be worth considering. Such an approach might provide a period of automatic anti-cloning protection and an opportunity to register innovative software compilations or subcompilations in order to get a longer period of blockage or a period of compensation under a standard licensing arrangement. A software developer might register, for example, a new user interface design, a macro language, a new algorithm, or the like, without having to register the product as a whole, as is required under SCPA. This approach may best match the design principles set forth above.

As with SCPA,[429] registration should probably be required within a year or two of the first commercial distribution of a product embodying the legal regime.[430] As with SCPA,[431] the regime should employ a light, copyright-like registration process, rather than a patent-like examination process, although it should allow later opportunities to challenge whether the registered material qualifies for protection.[432] This would

protection ceases. The Register of Copyrights is empowered to prescribe a form for registration of mask works. See 17 U.S.C. § 908(c) (1988). "Form MW" is designated for use by registrants for protection under SCPA. See 37 C.F.R. § 211.4(b) (1993). Block 6 of this form requires disclosure of the date of first commercial exploitation of the mask work. See Form MW, reprinted in Richard H. Stern, Semiconductor Chip Protection § 3.5 (1987). Masks deposited with the Copyright Office also make it easy to know with certainty the exact protected design. Finally, registration also makes it easier to record transfers of intellectual property rights in chip designs. See 17 U.S.C. § 903 (1988).

428. See supra note 88–91.

429. See supra note 419 and accompanying text.

430. Prompt registration produces an objective evidentiary record of the identity of the mask work as of the date of registration. See Stern, supra note 427, § 3.1.

431. See supra note 420 and accompanying text.

432. This might be done in litigation, as is the case in copyright law, see, e.g., Alfred Bell & Co. v. Catalda Fine Arts, Inc., 191 F.2d 99 (2d Cir. 1951) (defendant challenging originality of plaintiff's work in infringement litigation), or by permitting opposition to registration application, as is common in trademark law, see 15 U.S.C. § 1063 (1988).

minimize the up-front costs associated with gaining protection, in accordance with the market-oriented principles discussed above.[433]

As with SCPA, registration might bring with it a longer period of blockage, or alternatively, an automatic royalty-bearing license available on standard terms after expiration of the unregistered design protection right.[434] Of the two alternatives, we favor the latter because it would remove the transaction costs of licensing. Reasonable fixed fees would encourage second comers to compensate the innovator rather than duplicating effort. They would also prevent the innovator from getting hit by its own hardball in licensing negotiations.

7.6 A Market Segment Approach

Exclusive rights regimes do not generally link the scope or duration of protection available to a qualifying innovation with the market proximity of a second comer's product.[435] However, a market-oriented legal regime for protection of industrial designs in software might do so. That is, a second comer's ability to enter the market might be regulated according to how close the second comer's market is to the innovator's market.

In prior work, Professor Reichman has pointed out that crafting an appropriate scope for derivative work rights in an industrial property law is a very difficult problem.[436] Granting very broad derivative work rights can unduly impede competition in the general products marketplace. On the other hand, providing no means for control over derivative uses of a technical innovation may keep investment incentives below optimal levels. Regulating use of the innovation by market segment might provide a mechanism with which to achieve the needed balance.

An innovator's most immediate concern is to enjoy the advantage of its innovation in the market for its product. A second comer in the same market threatens the first comer's lead time advantage in a much more immediate manner than a second comer in an adjacent market. However, once it has established a niche in the market for its first product, an

433. See supra note 411 and accompanying text.

434. See infra notes 486–487 and accompanying text. An electronic repository of registered materials might, in a networked environment, accomplish licensing on standard terms electronically, thereby minimizing transaction costs. Section 8.3.2 will give further consideration to how electronic repositories of software and software components already in existence on the Internet might evolve to become exchanges for transactions in registered software components.

435. Trademark law takes market segment into account in that goods bearing the name of a first comer's product are generally regarded as noninfringing of trademark rights if the second comer's product operates in a market remote from that of the first user. See, e.g., McGregor-Doniger Inc. v. Drizzle Inc., 599 F.2d 1126, 1134–35 (2d Cir. 1979) (finding no infringement between expensive women's coats and inexpensive golf jackets).

436. See Reichman, Electronic Information Tools, supra note 5, at 457–58 (arguing that copyright law can control too much derivative use and traditional rules governing industrial tools too little).

innovator may perceive opportunities to reuse the innovation in adjacent markets. Thus, a second comer's adoption of the innovation in an adjacent market would have some potential to undermine the innovator's ability to use the innovation in adjacent markets. Because of this, some regulation of adjacent markets may be warranted. If, however, the second comer's market is very distant from the innovator's market, the innovator is unlikely to look to it as a way of recouping its research and development expenses. Consequently, the second comer's use of the innovation in that market may not have market-destructive effects and should be lightly regulated, if at all. Derivative uses of the innovation in remote markets might be blocked for a shorter period of time, or might be subject to an automatic license rather than a lead time blocking period.

Market segment might also affect the degree of similarity required to trigger liability. When an innovation in software migrates to a remote market, it will often need to be adapted to its new environment and may not be identical or substantially identical to the embodiment in the innovator's product even if it accomplishes exactly the same behavior.[437] An adjustment in scope of protection might be needed to take account of differences attributable to the different domain, assuming one desired to give an innovator some opportunity to be the first to take the innovation to remote markets or receive some compensation when such a migration occurred.

This approach would satisfy some of our market-oriented design principles better than the general anti-cloning automatic protection regime discussed above. Yet, such a graded protection scheme might be difficult to implement. A market segment approach would be more uncertain than a standard blockage term, independent of market segment.[438] There are, however, some successful legal regimes that base important legal distinctions on market segment concepts.[439] Market proximity may be an appropriate factor to consider in setting standardized royalties for reuse of a registered innovation, especially if industry-wide blanket licenses emerge to implement a liability regime.

437. See supra, text accompanying notes 382-383.

438. Products do not always stay in their original markets. The Sideways program, for instance, was first introduced in an adjacent market to Lotus 1-2-3, but was later incorporated into the Lotus product.

439. Unfair competition law has often taken market segment into account. See, e.g., International News Serv. v. Associated Press, 248 U.S. 215, 239-40 (1918) (applying different rule to use by direct competitor than to use by public); United States Golf Ass'n v. St. Andrews Sys., Data-Max, Inc., 749 F.2d 1028, 1038-40 (3d Cir. 1984) (finding appropriation by indirect competitor lawful). The same is true for trademark and dilution law. See, e.g., Mead Data Cent., Inc. v. Toyota Motor Sales, U.S.A., Inc., 875 F.2d 1026, 1031-32 (2d Cir. 1989) (holding that "Lexus" automobiles will not harm "LEXIS" trademark legal database). Antitrust laws also make distinctions on the basis of market segment.

7.7 An Improvements-Oriented Approach

We also considered an approach that would differentiate between those who made improvements to an innovative program compilation derived from another firm's product and those who imitated the compilation without making improvements. One might, for example, let an improver come to market sooner than one who copied an equal quantum but made no improvements. Alternatively, one could block the copier while allowing the improver to license for a standardized fee.

A reason to distinguish between them is that imitations generally only increase the supply of products embodying the innovation and contribute to price competition. Improvements, on the other hand, benefit the market because they elevate to some degree the overall level of skill in the technical community.[440]

As indicated above, giving consideration to improvements would probably be desirable if a substantial similarity standard, rather than a substantial identity standard, was selected as the standard by which to judge when a second comer had unfairly interfered with the market opportunities of an innovative software developer.[441] However, although it is often easy to tell that a second program improves on the original,[442] in some instances it can be difficult to say whether differences from the original are improvements or merely attempts to avoid liability.[443] Of course, the market will generally determine which differences are improvements: consumers will favor an improved version. This measure, however, does not easily distinguish between substantive improvements and the price improvement that all clones bring to the market. However, the fact that improvements are not readily distinguishable from mere differences suggests that this standard could be somewhat troublesome.[444]

8 Alternative Courses of Action

Policymakers have at least three near-term options regarding legal protection to computer program innovations. One is to do nothing and let the cycles of under- and overprotection from existing legal regimes play themselves out. These cycles will be harmful to the software industry and consumers, but the industry may well be strong enough to weather

440. See, e.g., Reichman, Legal Hybrids, supra note 100, at 2539–40, 2548–50.

441. See supra notes 375–377 and accompanying text.

442. Lotus 1-2-3, for example, was a significant improvement over VisiCalc. See, e.g., Barr, supra note 243, at 169.

443. For example, it is unclear whether to regard Borland's display of the Lotus command hierarchy by pull-down menu, rather than by a two-line moving cursor menu at the top of a screen, as an improvement or a difference for difference's sake.

444. On the other hand, we observe that SCPA takes improvements (or lack of improvements) into account in judging whether a similar semiconductor design should be considered an infringement. A second comer who makes improvements to a semiconductor design is unlikely to be found an infringer unless its chip is substantially identical to the original. See supra note 382 and accompanying text.

the law's inadequacies. A second is to make minimal changes in the law to address the most pressing underprotection problem facing software developers: the lack of protection against cloning of program behavior and other industrial design elements of software. The third option—which we recommend—would give serious consideration to establishing a registration-based system that would complement anti-cloning protection to ensure that innovators would receive contributions from those who wish to reimplement their innovations.

8.1 Doing Nothing

While doing nothing is not the option we recommend, it is worth noting that existing law does provide meaningful protection to some commercially valuable aspects of programs. Copyright protects against the most market-destructive appropriations of program behavior, i.e., exact duplication of code.[445] Copyright also protects against appropriation of expressive displays of text and graphics produced when program instructions are executed.[446] Patent and trade secrecy law provide some important supplemental protection to software developers.[447]

It is understandable that many are satisfied with results achieved under existing legal regimes. The software industry is highly competitive and innovative, both domestically and internationally. Moreover, it is also growing rapidly. Proponents of the status quo suggest that existing legal regimes be given additional time to evolve. If experience proves that more legal protection is needed for programs, they say, there will be time to pass it then. Those who have only recently finished the very arduous struggle to gain international consensus on legal protection for computer programs through copyright law would like to believe that the decades-old debate on this subject is at long last definitively resolved.

Our response is two-fold. First, we have shown that existing legal regimes cannot evolve to provide appropriate protection for software innovations because existing regimes and the valuable innovations embodied in programs are fundamentally mismatched.[448] Second, we have shown that the present equilibrium is more fragile than it seems. United States policymakers revealed how fragile they consider the equilibrium to be by their sharply negative response to the recently proposed amendment to the Japanese copyright law that would have permitted decompilation of software (a proposal, ironically enough, consistent with United States case law developments). Both the United States Trade Representative and officials of the Commerce Department complained that the change would permit Japanese companies to quickly reproduce software

445. See supra notes 285–286 and accompanying text.

446. See supra notes 164–171 and accompanying text.

447. See supra notes 88–90 (trade secrecy), 220 (patents) and accompanying text.

448. See supra notes 117–189 and accompanying text.

that they would be unable to write independently.[449] United States officials reacted similarly when the European Council was considering its directive on the legal protection of computer programs.[450] Furthermore, several factors may contribute to a destabilization of the current situation: (1) the current trend in court decisions in software copyright cases of filtering out valuable industrial design elements of programs may embolden cloners to re-enter the software market;[451] (2) decompilation technology may improve enough to make it trivially easy for competitors to acquire the industrial design of program internals that today can, for the most part, be kept secret;[452] and (3) the issuance of a large number of questionable patents for software-related ideas may impede competitive development and follow-on innovation in the software industry.[453] Moreover, the trend in United States case law, giving programs a very "thin" scope of copyright protection, could influence the interpretation of copyright protection for programs on a worldwide basis. If so, the United States may, in the not too distant future, be leading an effort to gain another international consensus about software protection.[454] In anticipation of this possibility, the Article aims to start a discourse about a proper framework for providing additional legal protection to software innovations.

8.2 The Minimal Change Option: Anti-Cloning Protection

Because of their vulnerability to trivial acquisitions of equivalence, externally discernible compilations of applied program know-how, such as behavior and user interfaces, may need some artificial lead time that classical intellectual property regimes do not provide. Although development cycles for programs provide some natural lead time to those who introduce innovative behavior and user interfaces to the market, this lead time may be inadequate to the needs of the industry.[455] These compilations should be protected from commercial appropriations that are destructive of lead time, including protection against clones, near-clones, and partial clones. Although the need for legal protection against the cloning of internal elements may be less immediate in view of the current

449. See, e.g., T. R. Reid, A Software Fight's Blurred Battle Lines: U.S. Computer Companies Are on Both Sides as Japan Considers Copyright Law Changes, Wash. Post, Jan. 11, 1994, at D1. Pressure from the United States was so strong that the Japanese proposal was dropped.

450. See Vinje, supra note 328.

451. See supra notes 204–209 and accompanying text.

452. See supra notes 111–116 and accompanying text.

453. See supra notes 221–225 and accompanying text.

454. United States trade negotiators cannot reasonably expect to persuade other nations to construe copyright for programs more broadly than their own courts do. Thus, if supplemental protection is needed for software, a new consensus will have to be cultivated in the international arena.

455. See supra notes 233–241 and accompanying text.

difficulties of decompilation,[456] we recommend taking a more forward-looking approach and regulating these compilations under the same anti-cloning approach.

Because a market-oriented legal regime for a fast-moving technology like computer software should minimize bureaucracy, protection against cloning should arise automatically by operation of law from the first commercial distribution (or public exhibition) of the program embodying it.[457] Protection against cloning should last long enough to give developers sufficient lead time to develop a niche in the market, but not so long as to impede the incremental development of technology or the creation of new de facto standards in the marketplace.[458]

We note that a number of countries have recently adopted or proposed short terms of protection (generally three years or less) for unregistered designs of industrial products.[459] A concept of this sort might usefully be adopted for protecting program compilations, at least insofar as there is some minimal creativity in the compilation (that is, it does not consist entirely of standard or commonplace elements arranged in a standard or commonplace way).[460]

Anti-cloning protection of the sort we recommend for programs could be implemented by legislation or by common law. In other countries, unfair competition law has served as a breeding ground for new forms of intellectual property protection.[461] The misappropriation branch of unfair competition law, particularly as manifested in *International News Service v. Associated Press* (*INS*),[462] is a possible foundation on which a common-law form of anti-cloning protection for software could be built.

INS, oddly enough, anticipates the critical legal protection problem of information-based economies: the vulnerability of costly-to-develop and commercially valuable information products to rapid, market-de-

456. See supra note 204 and accompanying text.

457. See supra section 7.2.

458. See supra notes 368–371 and accompanying text.

459. See Reichman, Legal Hybrids, supra note 100, at 2464–65.

460. Like SCPA, an automatic anti-cloning regime for software innovation should not provide protection to designs that are staple, commonplace, familiar, or minor variations on staple designs. See 17 U.S.C. § 902(b) (1988).

461. See Walter J. Derenberg, The Influence of the French Code Civil on the Modern Law of Unfair Competition, 4 Am. J. Comp. L. 1, 3–4 (1955). Prior to the amendments to the copyright statute that added sound recordings to the subject matter of copyright, unfair competition law also protected sound recordings in the United States. See Howard B. Abrams, Copyright, Misappropriation, and Preemption: Constitutional and Statutory Limits of State Law Protection, 1983 Sup. Ct. Rev. 509, 518–24 (discussing early cases applying misappropriation doctrine to sound recordings).

462. 248 U.S. 215 (1918). For an effort to modernize and reconstruct the *INS* misappropriation tort as an appropriately limited legal doctrine based on restitution principles, see Gordon, supra note 101, at 266–73.

structive copying.[463] Associated Press brought suit against International News Service because International News Service was appropriating news from early editions of Associated Press-affiliated newspapers and publishing it in their own affiliated newspapers that competed directly with the Associated Press papers.[464] To preserve incentives for Associated Press to invest in news-gathering, the Supreme Court decided to give it lead time protection in the commercial distribution of the news.[465] Software developers, too, need some lead time before competitors can appropriate compiled know-how from their products. It is conceivable that the common law could evolve anti-cloning rules for software innovations, employing market-oriented principles akin to those applied in *INS*,[466] and refined to reflect the primary dimensions set forth in Section 5. Software developers might, however, prefer to go directly to Congress to get statutory anti-cloning protection.

If the United States adopts anti-cloning rules for software, this experience could lay the groundwork for reaching an eventual international consensus that could either be subsumed within the Paris Convention under article 10*bis*, adopted by a GATT/TRIPs panel as a norm of international trade, or included in a revision of the GATT/TRIPs agreement.[467] One of the lessons of the twentieth century has been that legal

463. In recognition of the market-destructive potential for misappropriations of data in electronic databases, the European Council is considering adoption of a directive to protect data against unfair extractions. See Amended Proposal for a Council Directive on the Legal Protection of Databases, 1993 O.J. (C 308) 1, 8.

464. Associated Press (AP) news could not be protected as a trade secret once it had been published in early editions of AP newspapers or posted on public bulletin boards in AP offices. Nor could it be protected by copyright law because, at the time, AP papers did not copyright their papers. Even if they had, that would not have protected the news as such from appropriation because news is beyond the scope of copyright. See *INS*, 248 U.S. at 233–35.

465. The Court enjoined International News Service from misappropriation of AP news until the commercial value of the news had passed away. See id. at 245–46. International News Service could, however, use AP news to get "tips" on newsworthy stories. See id. at 243–44. Courts employing market-oriented principles in software disputes might also look to cases interpreting non-competition agreements and limiting them to a reasonable duration or scope as useful precedents. See, e.g., Telxon Corp. v. Hoffman, 720 F. Supp. 657, 665–66 (N.D. Ill. 1989) (discussing court discretion to "blue pencil" terms when noncompetition agreements are overbroad).

466. The *Sears-Compco* tradition favoring free competition as to unpatented, uncopyrighted innovation has, however, historically limited judicial developments of this kind. See Sears, Roebuck & Co. v. Stiffel Co., 376 U.S. 225 (1964); Compco Corp. v. Day-Brite Lighting, Inc., 376 U.S. 234 (1964) (striking down the use of state unfair competition law to protect unpatentable, uncopyrightable designs). The Supreme Court recently reaffirmed the tradition, striking down a Florida statute that forbade use of plugmolding processes in the manufacture of boats and their component parts. See Bonito Boats, Inc. v. Thunder Craft Boats, Inc., 489 U.S. 141 (1989).

467. Reichman predicts that GATT/TRIPs will need to be amended to deal with a number of scope of protection issues, as well as with issues that today tend to be dealt with through legal hybrid regimes, such as the EU's proposed database directive, see supra note 463, and industrial designs. See Reichman, GATT/TRIPs, supra note 146, at 173–81. See

institutions, including intellectual property laws, must adapt to shifts in economic structure, such as the transition from largely manufacturing-based economies to information economies.

In the event Congress or the courts devise an appropriate market-oriented legal regime to protect software innovation, policymakers should also reassess the need for software patents. The recent interest of some software developers in patents may arise less from a conviction that patents are well-suited to the industry's needs, than from a sense that copyright alone may not provide sufficient protection.[468] We believe that the software industry would favor the market-oriented, lead time approach we propose because it matches so well the environment in which the industry has thus far innovated and prospered.[469]

If policymakers decide that patents should have a role in the protection of software innovations, they should encourage continuation of the reforms that the Patent and Trademark Office has undertaken to remedy the deficiencies in its standards and processes for dealing with software innovations.[470] In addition, Congress should amend the patent law to require publication of patent applications within eighteen months of filing. Prompt publication may prevent the issuance of patents for software ideas that are already in the prior art or make only minor advances upon it.[471] Congress should also consider opening up the reexamination process.[472] Software developers in possession of prior art that they think defeats a patent should be able to do more than send the information to the Patent and Trademark Office with a humble request for reexamination. These reforms and legislative actions are needed in order for the software industry to gain confidence in the soundness of the patent system as applied to software.[473]

also Paul E. Geller, Intellectual Property in the Global Marketplace: Impact of TRIPS Dispute Settlement?, 28 Int'l Law. 1, 14–15 (1994) (discussing Article 10*bis* of Paris Convention and possibility that GATT/TRIPs panel might decide that certain forms of unfair competition have become norms of international trade within scope of current GATT/TRIPs agreement).

468. See supra notes 222–225, 302 and accompanying text.

469. See supra notes 242–246 and accompanying text.

470. See supra note 224 and accompanying text.

471. See S. 1524, 103d Cong., 2d Sess. (1994) (Patent Simplification Act of 1994).

472. See 35 U.S.C. §§ 301, 307 (1988). These provisions permit someone who has information about patents or printed publications bearing on the validity of an issued patent to request a reexamination of the patent in light of the cited prior art. If the Commissioner of Patents finds that no substantial question of patentability exists, he or she will deny the request for reexamination. This decision is final and unappealable. If the Commissioner decides that such a substantial question does exist, the patent owner can provide a written statement in support of the patent. The person requesting the reexamination may then respond to this statement. Although the Patent and Trademark Office can hold a hearing on a reexamination matter, the statute does not contemplate an opportunity for the requesting party to participate. The United States has recently committed itself to broadening its reexamination process. See, e.g., Teresa Riordan, U.S., Japan In Accord on Patents, N.Y. Times, Aug. 17, 1994, at D1, D18.

473. See supra notes 134, 221–223 and accompanying text.

8.3 A More Forward-Looking Approach: Investigation of a Registration and Automatic Licensing System for Software Innovations

Although anti-cloning legislation will take care of the industry's most pressing protection need, it is only a partial solution. A broader solution would include adoption of a registration system for innovative compilations of applied know-how embodied in software. This system would facilitate development of a repository of state-of-the-art software that would be of assistance in the evolution of software engineering, as well as potentially provide an exchange through which low-cost transactions about reuse of software innovations could occur. A registration system of this sort might also provide incentives for software developers to disclose high-level descriptions of innovative algorithms and other internal design elements of software that today are kept secret. Registration would provide a way for innovators to get compensation for disclosed innovations without undergoing the lengthy, expensive, and cumbersome process of patenting.

One of the challenges is to conceive a legal framework that would be adaptable as the technology and markets for software evolve. The next subsection raises a number of future-directed issues.

8.3.1 Evolution of Technology May Affect the Legal Landscape

Future technological developments will almost certainly disrupt whatever equilibrium exists today or develops in the near term. Some developments may make it easier for second comers to acquire behavioral equivalence with an innovator's software product. Insofar as this occurs, there will be a need to adjust the legal regime to ensure that market-destructive appropriations do not go unremedied.

Improvements in technological tools for software development[474] may, for example, so speed up development cycles as to force adjustment of the lead time protection period for some aspects of software, such as features, that under current conditions may have relatively adequate natural lead time.

Decompilation technology may also advance to the point that all of the know-how embodied in program internals is as easily discerned as if it were published on the surface of the product.[475] If this occurs, the rules governing decompilation, as well as innovations in interfaces, algorithms, and other internal design elements of programs, might need to be rethought to ensure adequate lead time protection for them.

474. See, e.g., Gene Forte & Ronald J. Norman, A Self-Assessment by the Software Engineering Community, Comm. ACM, April 1992, at 28 (discussing goals of computer-aided software engineering).

475. See supra notes 114–116 and accompanying text.

Technological means for protecting digitized intellectual property are also likely to improve.[476] One positive result of this development will be to lessen the need for the law to protect the innovations they embody. But technological developments for protecting intellectual products may have unsettling effects on the legal equilibrium as well. Such technology might, for example, make any legal rules about decompilation as a means of gaining access to internal program know-how obsolete.[477] The challenge could come to be how to induce firms to license internal program know-how, especially interface information, so that competition in development of interoperable systems could continue without socially wasteful duplications of effort.[478]

Technology may also affect the nature of software markets. Technologies for controlled access to computer programs through networks may, for example, open up markets for computer programs that are not based on the distribution of copies (which is what copyright law knows how to regulate), but rather controlled access to executable uses of them.[479] A market-oriented legal regime for programs might provide default rules for these transactions as well.

8.3.2 Electronic Repositories Can Become Software Exchanges

Evolution of software technology may open up a number of opportunities for electronic markets, with very low transaction overhead, as a means for distribution of and billing for commercial uses of software innovations.[480] Under a system that provided a low-transaction-cost means for obtaining compensation, innovators in software would have incentives to favor reuse of their innovations,[481] because they would be guaranteed revenue for each reuse.

Electronic markets for software may evolve out of electronic repositories of reusable software components that already exist.[482] Some firms

476. See, e.g., Proceedings, Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment, J. Interactive Multimedia Ass'n, Jan. 1994 [hereinafter Technology Proceedings].

477. It might, for example, thwart the intent of the European software directive that forbids contractual restrictions on decompiling necessary to achieve interoperability. See EC Directive, supra note 7, at art. 6.

478. See supra Sections 6.1.7 and 6.1.13.

479. See Marvin A. Sirbu, Internet Billing Service Design and Prototype Implementation, *in* Technology Proceedings, supra note 476, at 67.

480. A number of commentators have begun to see that digital networks may make automatic licensing possible. See, e.g., Zentaro Kitigawa, Copymart: A New Concept—An Application of Digital Technology to the Collective Management of Copyright, *in* WIPO Worldwide Symposium on the Impact of Digital Technology on Copyright and Neighboring Rights 189 (1993) (suggesting network markets in both copies and copyrights). See also Finding a Balance, supra note 2, at 161–79 (discussing difficulties attending protection of digital information).

481. See, e.g., Sirbu, supra note 479.

482. In April 1991, the National Program for High Performance Computing and Communications sponsored a workshop entitled "Toward a National Software Exchange"

have developed repositories of reusable code for particular functions.[483] Some have developed electronic repositories of more abstract components of software, such as algorithms.[484] Some reuse repositories have been developed by private firms for internal use only; others have been developed with government funds and are more widely accessible.[485] Further growth of these kinds of repositories is foreseeable as digital networks expand.

The electronic repositories of algorithms that already exist on the Internet tend to be libraries of publicly known algorithms. Users of these libraries can both browse and download algorithms from them.[486] These repositories could also evolve into software exchanges. If this occurred, those who wanted to be compensated for commercial uses of their innovative algorithms might register them with the repository that could then make them available on an automated licensing basis.

The developer of an innovative algorithm could, for example, send the algorithm to an electronic repository. The repository could run a search of its collection (and perhaps those of other repositories) to determine if the algorithm was already present.[487] If not, the registration would be accepted, subject to defeasance if representations made at registration were later determined to be false. The algorithm could then be indexed and added to the repository.

---

that aimed to assist existing and future electronic repositories of software and software components to become software exchanges. See Michael Raugh, NASA Ames Research Center, Examples of Service Components for a National Software Exchange: A Position Statement (1991) (on file with the Columbia Law Review).

483. See Bruce D. Nordwall, Reusable Software Can Reap Big Savings in Avionics, Aviation Wk. & Space Tech., Nov. 16, 1992, at 51; Software Reuse Library System, Defense Elec., Oct. 1992, at 16.

484. See Neil A. Maiden & Alistair G. Sutcliffe, Exploiting Reusable Specifications Through Analogy, Comm. ACM, April 1992, at 55.

485. See, e.g., James Baldo & Craig A. Will, Strategy and Mechanisms For Encouraging Reuse in the Acquisition of Strategic Defense Initiative Software, Institute for Defense Analyses Paper P-2494, at vii (June 1990) (reporting that most software reuse occurs within firms).

486. Netlib is an electronic repository of algorithms established at the Oak Ridge National Laboratory. It can be accessed through Internet at netlib ornl.gov. See Jack Dongarra, Lessons Learned from Netlib, Address Before the NASA Ames Research Center Conference: Towards a National Software Exchange (Apr. 10, 1991). For access information, see Richard Morin, More Math Madness, UNIX Rev., May 1994, at 93.

487. We envision electronic software component repositories as private businesses, rather than as part of a government operation. We expect they would operate very much in the manner that copyright collectives do today in setting standard terms and conditions for certain uses of items in the repertoire. To maintain a central national registration system, there might still be a government-run entity to which innovators might apply directly or through the repository which they choose as a licensing agent. Incentives to use the repository as a licensing agent might arise from a desire to avoid the licensing regime that the government might impose on those not making other arrangements, as happens now with "Harry Fox licenses." See supra note 249.

One possible licensing procedure would be to allow registrants to set their own terms. Another would be an advisory body to create a standardized license. A third might be to have some period in which the registrant could set the terms, and a later period in which the repository's standard terms would apply. Upon expiration of the automated license term, the algorithm would become a component of the public library, available for use without restriction. Some users of the repository might subscribe only to the public library; others might subscribe to the commercial exchange. Subscription levels could be geared to the desired level of use.

A system of this sort would have the advantage of enhancing the public's access to algorithms, adding to the storehouse of knowledge available to software engineers, and providing a means for innovators to get compensation. We do not underestimate the complexities of developing an appropriate set of terms or duration for such a licensing scheme. Neither do we view this as an insuperable obstacle, however, if the will exists to envision and create new types of markets for software innovations. We note that the music industry has evolved standard licensing arrangements that, to some extent, parallel the standard licensing arrangements we suggest for software.[488]

Conclusion

The principal goal of this Article has been to make manifest certain important attributes of computer programs, to clarify how they are developed, and to explore what kind of industry and market has emerged for their distribution. We feel that such an elucidation is essential for making appropriate judgments about legal protection for computer programs.

Existing law has principally focused on only one characteristic of programs: their texts, that is, the statements and instructions that can be used in a computer to bring about certain results. Virtually all nations have recognized the textual character of program code in deciding to use copyright law to protect it. Copyright is proving to be an effective and uncontroversial means for protecting program code from exact duplication.

Existing legal regimes have, however, found it difficult to recognize and deal appropriately with other characteristics of computer programs. Our review of some of the prominent software copyright cases shows that the real issues in controversy have sometimes been obscured by the issues that copyright doctrine seems to demand. We contend that the real dispute in the "look and feel" cases, for example, was not about the arrange-

488. See Bernard Korman & I. Fred Koenigsberg, Performing Rights in Music and Performing Rights Societies, 33 J. Copyright Soc'y 332 (1986); see also Finding a Balance, supra note 2, at 36 (suggesting that new copyright collecting entities may be needed for digital information products).

ment of words, icons, or the like—elements of user interfaces that have doctrinal significance in copyright law—but about the lawfulness of developing a program that imitates another program's behavior.

We understand the temptation to use copyright law to protect behavior. Behavior is the most valuable aspect of programs; its development is costly and intellectually demanding. It can be rapidly imitated by someone who does not incur the same level of research and development costs as the innovator because the know-how embodied in it is borne on the face of the software product. A skilled engineer need only study the program in operation to know how to create a functionally indistinguishable product. There might well be too little incentive to develop innovative program behavior if clone products could appear very quickly on the market. However, protection of useful program behavior by copyright law contravenes both its traditional principles and explicit provisions of the statute that embodies them. Copyright protection would also provide such a long period of protection to useful behavior as to bring about market-destructive effects of a different sort, harming consumer welfare by banning for seventy-five years functionally indistinguishable products, having independently created texts.

Similarly, the controversy over decompilation is not really about an intermediate copy of the program code that occurs in the decompilation process. The focus on intermediate copying derives from the fact that copying is doctrinally significant in copyright law. The real concern is that valuable internal design elements that can be discerned through decompilation (such as algorithms and information necessary to permit programs to interoperate) are potentially vulnerable to rapid copying that patent and copyright cannot effectively rectify or prevent, so long as the imitator independently develops a program with a different text.

Here, too, we understand the temptation to use copyright law to stop potentially market-destructive appropriations of the know-how borne near the surface of software products, i.e., in the industrial design that produces efficient behavior. Once revealed by decompilation, the compiled know-how in internal program design cannot be effectively protected by trade secrecy, copyright, or patent law. If copyright law were to be used to prevent decompilation, the uncopyrightable, unpatentable know-how embodied in program internals could be kept as a trade secret. But this would transform copyright into a trade secrecy law. Courts and policymakers have rightly recognized the overprotective potential of using copyright law to obtain de facto monopolies to unpatentable technical innovation.

We propose to remedy market-destructive appropriations of program behavior and the industrial designs aimed at producing efficient program behavior through a period of automatic anti-cloning protection for these innovations. The period should be long enough to give efficient incentives to invest in the development of innovative software, yet short enough to avert the market failure that would result if second comers and follow-on innovators were blocked from entering the market long after the first firm had recouped its initial investment. We have not attempted to fix a precise duration for this period or other implementation details that need further discussion and debate. We aim to provide a framework for determining what constitutes a clone and when cloning would be likely to have market-destructive effects. We also propose further study to develop a registration system by which innovative components of programs might become eligible, before expiration of the automatic anti-cloning period, for an additional period of compensation for use of their innovations on standard terms and conditions of the sort that copyright collectives today employ for subject matters they license.