COMPUTER ETHICS

Deborah G. Johnson
Georgia Institute of Technology

Prentice Hall

Upper Saddle River, New Jersey 07458

10. Describe three policy approaches to protecting personal privacy. What are the advantages and disadvantages of each?
11. How does globalization of the economy impact information gathering practices?

SUGGESTED FURTHER READING

AGRE, PHILIP E., and MARC ROTENBERG, *Technology and Privacy: The New Landscape* (Cambridge, MA: MIT Press, 1997).

BENNETT, COLIN J., and REBECCA GRANT, *Visions of Privacy, Policy Choices for the Digital Age* (Toronto, Ontario: University of Toronto Press, 1999).

DE CEW, JUDITH WAGNER, *In Pursuit of Privacy, Law, Ethics, and the Rise of Technology* (Ithaca, New York: Cornell University Press, 1997).

REGAN, PRISCILLA M., *Legislating Privacy, Technology, Social Values and Public Policy* (Chapel Hill: University of North Carolina Press, 1995).

SMITH, ROBERT ELLIS, "Compilation of State and Federal Privacy Laws," *Privacy Journal* (2000).

WEB SITES

For the Electronic Privacy Information Center, see: www.epic.org
For Privacy International, see: www.privacyinternational.org/

CHAPTER 6

Property Rights in Computer Software

SCENARIO 6.1 Obtaining pirated software abroad.

Carol works as a computer consultant for a large consulting company. She loves her job because it allows her to continuously learn about new IT applications and she uses many of these applications to do her work. Carol has vacation time coming and she and her husband decide to spend two weeks visiting a country in Southeast Asia. While there, Carol and her husband decide to check out the computer stores to see what sort of software and hardware is available. While rummaging in one store, Carol finds an office suite package. It includes a spreadsheet, a word processor, presentation applications, and more. Indeed, it looks identical to a package made by a well-known American company, a package that she has been thinking about buying. The price in the U.S. is around $1,200 which is why she has been reluctant to buy it, but here the package costs the equivalent of $50. She has heard that countries like the one she is in do not honor U.S. copyrights. She notices that the written documentation looks like a duplicated copy; it doesn't look like it has been professionally printed. The deal is just too good to resist. She buys the software!

As she prepares for the airplane trip home, she wonders where she should put the package. She's not sure what will happen if custom officials notice it as she re-enters the United States.

Has Carol done anything wrong? Would it be unfair if U.S. custom officials stopped Carol and confiscated the software package?

SCENARIO 6.2 Stealing the idea.

Imagine that it is 1980. Use of computers is growing and there are a wide variety of computers and operating systems available. Bingo Software Systems has an idea for a new operating system that, it believes, will be significantly better than anything that has yet been produced. Bingo is a small company employing twenty people. It obtains venture capital and spends three years developing the

operating system. Over the course of the three years, Bingo invests approximately two million dollars in development of the system. When completed, the new system is successfully marketed for about a year, and Bingo recovers about 25 percent of its investment. However, after the first year, several things happen that substantially cut into sales. First, a competing company, Pirate Pete's Software, starts to sell a system very similar to Bingo's, but Pirate Pete's system has several features not available in Bingo's system. It appears that Pirate Pete's Software has examined the system developed by Bingo, adopted the same general approach, possibly copied much of the code, and, in any case, produced a modified and improved version. Second, copying of Bingo's system is rampant. Customers, primarily small businesses, appear to be buying one copy of Bingo's system, and then making multiple copies for internal use. It appears that they are giving copies to other businesses as well. As a result, there is a significant reduction in demand for Bingo's system. Bingo is unable to recover the full costs of developing the system and goes into bankruptcy.

Is this situation unfair? Has Pirate Pete's Software wronged Bingo Software Systems?

SCENARIO 6.3 Improving software.

Earl Eniac spends a year developing a virus tester. He finally has it developed to a point where he is pleased with it. It detects all the viruses he has ever encountered and repairs them. Earl makes the tester available from his Web site; anyone who wants it can download it for free. He also publishes an article in a computer journal describing how it works. Jake Jasper downloads a copy from the Web site, reads the article, and figures out how the virus tester works. He thinks it is a clever, creative, and useful piece of software. He also sees how several small changes could be made to the tester, changes that would make it even better. Since the copy that Jake downloaded from the Web allows him access to the source code, Jake makes these changes to the program. He then sends the revised program to Earl with an explanation of what he has done. Jake makes the improved version available from his Web site where he also describes what he has done, giving credit to Earl for the original.

Has Earl or Jake done anything wrong?

In this chapter, we turn our attention to a set of issues that arise because of the very nature of computer and information technology. In one sense, computers are simply machines. They are on par with other devices that we use to get things done—lawnmowers, washing machines, and automobiles. Many of these machines now contain computerized parts, but the point is that all machines are similar insofar as they are created by manipulating materials and laws of nature (e.g., laws of physics). In another sense, however, computer and information technology is different, and its distinctiveness becomes apparent when it comes to issues of ownership and property rights.

While it is difficult to pinpoint, the distinguishing feature of computer and information technology has to do with its intangibility and its reproducibility (discussed in Chapter 4). The standard way to conceptualize computer technology is to think of computers as a combination of hardware and software. *Hardware* refers to the machine, a malleable machine with practically infinite possible configurations. *Software* refers, essentially, to a set of instructions for the machine. Software controls and configures the machine. When software is put into a computer, it causes switches to be set and determines what the computer can and cannot do. The software sets up the computer so that it responds in determinate ways when users press keys or input programs.

Note that while this understanding of computers (as a combination of hardware and software) is fairly standard, the technology has evolved in ways that blur the distinction between hardware and software. For example, devices can be created by taking what is essentially a software program and etching it into a silicon chip. The microchip that is created is a hybrid between software and hardware.

When software was first created, it was considered a new type of entity. Nothing with the characteristics of software had existed before computers.[1] And, it is the ownership of this new entity that has posed daunting ethical and legal issues. To be sure, there have been and continue to be ownership issues with regard to computer hardware, but software has posed the most difficult challenge.

Software has created ethical and legal issues both at the macro or public policy level and at the micro or individual level. The broadest macro level issues have to do with whether or not, and what aspects of, computer software should be owned. In other words, what aspects, if any, of computer software should be legally protected as private property. The most compelling micro level issues have to do with whether it is wrong for an individual to make a seemingly illegal copy of computer software. For example, is it wrong for Mary in Scenario 1.1 (in Chapter 1) to copy the penny stock software? Or is it wrong for Carol in Scenario 6.1 to buy software that violates a copyright?

In the analysis that follows, I address both the legal and the moral issues and I address both descriptive and normative questions (i.e., what does current law specify and what should the law be in this domain). When it comes to the law, my focus is primarily on U.S. law though many countries of the world have similar copyright and patent laws.

The legal and moral issues are inextricably tied together in a number of ways. If laws set up unfair arrangements, then the moral critique of those arrangements is relevant to the interpretation of the law. Deciding what the

---

[1] Parallels have been made between software and the scrolls used in player pianos and between software and the keys used to set the functions of weaving looms. However, software is different from each of these in its form—digitalized instructions causing electronic switches to be set.

laws should be in a domain such as software ownership is in part a moral matter. For example, Scenario 6.2 suggests a problem in the legal protection provided to those who want to develop software, and this is also a moral matter in that it concerns the fairness of the situation and the social consequences of the situation. In an important sense, the law sets the rules of the game of software development, and the game ought to be fair in addition to producing good consequences. In addition to legal issues often being moral issues, moral issues often hinge on what the law is. For example, in Scenario 1.1, to determine whether Mary did something wrong, one important consideration is whether she broke the law. All else being equal, citizens have an obligation to obey the law. In Scenario 6.1, this question is more complicated since Carol is visiting another country in which the laws differ from those of her own country.

I begin this chapter by describing the legal mechanisms that are now available to those who develop and want to claim property rights in computer software. The question driving this description—a question on the minds of many in the software industry—is whether current legal tools are adequate for computer software. To fully understand the problems with software ownership, it is essential to take a broad philosophical perspective and consider the most basic question: Should computer software be owned at all? Should it be treated as property? You may find this question odd since so much is already owned in the domain of computing, by copyright or patent. Nevertheless, answering this philosophical question is important both for understanding why we have the laws that we currently have and for contemplating whether and how these laws might be changed in the future.

Software property rights continue to be uncertain in many areas and property rights claims are continuously being contested. The courts and law journals are filled with case after case in which this or that aspect of computing is being claimed as a property right—an encryption schema, a knowledge-discovery tool, a new operating system, a new digital-imaging technique, and so on. Moreover, the Internet has added several new dimensions to this already complex area. Search engines, privacy-enhancing tools, virus detectors, and other software systems are continuously being invented to implement activities on the Internet. In addition and consequently, international, cross-border property rights issues are intensified. While there have always been international law issues concerning property created in one country moving to another, the Internet has made these issues more intense and more pressing because it is so easy to send copyrighted materials and patented processes across national boundaries. International law must be brought to bear to determine the legal standing of something copyrighted or patented in one country when it moves to another country (Burk, 1994). International law is enormously complex, and there is no universally recognized power or authority to enforce international laws. The World Trade Organization is currently in the process of developing policies to cover intellectual property issues in international trade.

The property rights issues surrounding computer software are in many ways the most intriguing. Unfortunately, however, because there are so many issues, this chapter remains at the broadest level. The aim is to understand the foundations of property rights and then to explore how foundational principles apply to computer software. This analysis can easily be supplemented with current cases found in law journals, on the Web, or in newspapers.

After explaining the standard forms of legal protection currently available to software developers and identifying their limitations, I focus on the philosophical and moral foundations of property. I argue here that the moral arguments do not show that computer software *must* be treated as property. At least, neither utilitarian nor natural rights arguments establish that societies without intellectual property are unjust or immoral societies. Intellectual property rights are socially created rights and a variety of social arrangements are possible that are morally acceptable. Building on this analysis, I conclude the chapter with a discussion of the micro level question: Is it wrong for an individual to make a copy of proprietary software? Here I argue that it is wrong to make a copy because it is illegal, but not because there is some prelegal immorality involved in the act.

DEFINITIONS

To understand the software ownership issue, it is essential to understand what software is, and this requires understanding the distinction between algorithms, object programs (and code), and source programs (and code). As described earlier, software sets switches in computers and configures the computer in a specific way. Software sets up—programs—a computer so that it can receive and respond to commands given to it by a user or another program. What a particular program does to a computer can be described in three different ways: as an algorithm, in source code, and in object code. Though dated in the computer languages and technology to which it refers, Gemignani's 1980 explanation remains one of the most lucid:

> Programs are responses to problems to be solved. First, the problem in issue must be clearly formulated. Then a solution must be outlined. To be amenable to implementation on a computer, the solution must be expressible in a precise way as a series of steps to be carried out, each step being itself clearly defined. This is usually set forth as a flowchart, a stylized diagram showing the steps of the algorithm and their relationship to one another. Once a flowchart has been constructed, it is used as a guide for expressing the algorithm in a "language" that the computer can "understand." This "coding" of the program is almost certain to employ a "high level" computer language, such as BASIC or FORTRAN. When the algorithm is "coded" in a high level computer language, it is called a *source program*. A source program may bear a striking resemblance to a set of instructions expressed in literary form. The source program is fed into the computer by means of an input device, such as a terminal or card reader. The source program is "translated" by the compiler, a part of

> the operating systems program, into machine language, a language not at all similar to ordinary speech. The program expressed in machine language is called an *object program*. It is the object program which actuates the setting of switches which enables the computer to perform the underlying algorithm and solve the problem (Gernignani, 1980).

To update this account, we need only specify a somewhat different explanation of object and source code:

> Source code and object code refer to the "before" and "after" versions of a computer program. The source code consists of the programming statements that are created by a programmer with a text editor or a visual programming tool and then saved in a file. For example, a programmer using the C language types in a desired sequence of C language statements using a text editor and then saves them as a named file. This file is said to contain the source code. It is now ready to be compiled with a C compiler and the resulting output, the compiled file, is often referred to as object code. The object code file contains a sequence of instructions that the processor can understand but that is difficult for a human to read or modify. For this reason and because even debugged programs often need some later enhancement, the source code is the most permanent form of the program (see www.whatis.com).

As you will see later on, intellectual property laws treat each of these aspects of computer software in different ways.

THE PROBLEM

We can begin with the situation described in Scenario 6.2. Bingo Software is unable to sell its operating system software because Pirate Pete's has copied the software and made improvements on it. Pirate Pete's is able to sell its software at a markedly lower price because its development costs were so much lower than Bingo's. On the face of it, the situation seems unfair. The unfairness does not, however, arise simply from the fact that Bingo cannot recoup its investment. That happens frequently when a new business venture fails to find a sufficient market for its products. Rather the unfairness in the situation has to do with the fact that Pirate Pete's is able to use the work of Bingo without having to pay for it. Pirate Pete's is able to make a better system more cheaply by building on the work of Bingo. Bingo makes an investment in the development of a product with its money, its effort and its ingenuity, and yet gets none of the rewards. Pirate Pete's gets the reward without making the investment.

At first glance, this may seem a problem with an easy solution. All we have to do is give Bingo a legal right to own the software it creates, a legal right that excludes others from using or selling Bingo's software without Bingo's permission.

While this seems a good solution, it is much harder to implement than it seems, and there are good reasons for hesitating to give such protection. The

reasons for hesitating go to the heart of intellectual property law. In both copyright and patent law, there are limitations on what can be owned (claimed). In copyright law, expressions of ideas can be owned but not the ideas themselves. In patent law, applications or implementations of ideas can be owned, but abstract ideas, mathematical algorithms, mental steps, or laws of nature cannot be owned. Similarly, philosophical theories of property recognize that property rights are problematic when they get in the way of future development, creativity, and innovation. So, before leaping to the conclusion that we should give software developers the kind of ownership they need to avoid exploitation of the kind described in Scenario 6.2, we should consider the reasons for refusing to grant such protection.

CURRENT LEGAL PROTECTION

At present, three legal mechanisms can be used by those who create software to claim ownership in and protect their creations: (1) copyright, (2) trade secrecy, and (3) patent. In the late 1970s and early 1980s, a good deal of concern was beginning to be expressed that none of these legal mechanisms adequately protected computer software. That concern has persisted. In the 1970s and 1980s, a sizable literature described the extent and impact of software piracy and illegal copying and expressed fear that software development would be significantly impeded because software developers would not be able to recover the costs of development, let alone profit from their creations. This, it was feared, would significantly dampen the incentive to create.

While there is no indication that software piracy has subsided since the early 1980s, the environment for software development has changed. Many more patents on software-related inventions have been issued, and the software industry has found ways to successfully prosecute copyright infringement claims. Still, there is dissatisfaction with the legal situation. In addition to complaints about inadequate protection for software, there are serious concerns about too much being owned. The new concern is that copyrighting and patenting constrain software development because when a company develops a new piece of software, it has to do an extensive search to find out what is already claimed. Depending on what it finds, it may have to pay enormous fees to license a process that has been patented or it may have to craft a copyright claim in contorted ways to ensure that its claim does not infringe on others. All of this creates a barrier to new invention; it slows down the process and increases the costs of getting something to the marketplace.

Copyright

In the United States, when a software developer creates an original piece of software, the developer can use copyright law to obtain a form of ownership

that will exclude others from directly copying the software without permission. That is, others are not permitted to reproduce a copyrighted work, distribute copies of it, or display or perform the copyrighted work publicly, without first obtaining permission from the author (copyright holder). Until 1998, copyright protection extended for the life of the author plus 50 years. In 1998, copyright law was amended to extend the term of coverage to the life of the author plus 70 years.

Copyright protection is rooted in the United States Constitution where article I section 8, clause 8 specifies that Congress shall have the power "To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries." The Copyright Act protects "original works of authorship fixed in any tangible medium of expression, now known or later developed, from which they can be perceived, reproduced, or otherwise communicated, either directly or with the aid of a machine or device." (17 U.S.C. Section 102 (a) (1995))

While copyright provides a significant form of protection to authors, when it comes to computer software, the protection is limited and poses complex issues of interpretation. At the heart of copyright law is a distinction between ideas and expression. An idea cannot be copyrighted; the expression of an idea can be. The distinction between idea and expression makes sense when it comes to literary works. An author cannot own the ideas in his or her writing but can copyright the writing as a unique expression of those ideas. With this form of protection, authors are able to stop others from copying their writings without permission, and this makes it possible for authors to make money from the sale and distribution of their writings.

To be sure, copyright protection can be problematic for literary works. Claims to a number of famous literary works have been contested and authors have learned the hard way that the line between idea and expression is not so easy to draw. However, when it comes to computer software, the problems are even more complex because copyright does not seem to protect the most valuable part of software.

Generally, algorithms are thought to be the ideas expressed in a program and, hence, they are not copyrightable. When expressed as source and/or object code, however, programs are copyrightable. Both the source program and the object program are understood, in copyright law, to be "literary works," that is, formal expressions of ideas. The problem is that it is often sufficient for a competitor to study the software and with minor effort, and without directly copying, create comparable, and sometimes better, software. The task of creating new source and object code may be negligible once the algorithms or the ideas or the approach taken in the software is grasped from using the program.

In practical terms, this has meant a plethora of court cases and legal decisions trying to make clear when a copyright has and has not been violated. It may be fair to say that the extension of copyright to the domain of computer and information technology has created as many questions as answers.

The problem seems to be that the distinction between idea and expression is not suitable for software. Software is like literary works in being expressive, but it is unlike literary works in that it is also useful (functional). Software behaves; it performs tasks in a determinate way. The behavior of software is valuable and that value is not protected by copyright (Davis et al., 1996). As already mentioned, competitors can "read" the software and see and understand its useful behavior, and then create an original, alternative program behaving in the same way but not copying the literal expression of the original software.

Recognition of this difference between software and literary works has led some computer scientists and legal scholars to argue for a new and specially designed form of legal protection for software. Davis et al. (1996), for example, have argued for a new framework that protects against behavior clones for a limited period of time, and calls for a system of registration of software innovations but does not rule out the use of copyright for literal code. This system, they argue, would promote disclosure and dissemination while giving innovators time to develop their market.

In the absence of a new, alternative framework for protection, legal controversy surrounding current copyright laws persists with issues of interpretation being raised in a wide variety of ways. One important area of contention has been in determining when a copyright has been infringed. Whether or not there is infringement depends on whether the idea has been expressed in a new way or merely copies the pre-existing copyrighted expression. This determination is not easy to make.

When a software developer believes that her or his copyright has been infringed, the copyright holder must take the accused infringer to court. The burden of proof is on the copyright holder to prove infringement. To prove infringement, the copyright holder must show that there is a "striking resemblance" between the copyrighted software and the infringing software, a resemblance so close that it could only be explained by copying. If the defendant can establish that he or she developed the new program on his or her own, without any knowledge of the pre-existing program, then there has been no infringement. This is important to note: Copyright does not give a monopoly of control of a literary work. If someone else independently (without any knowledge of a pre-existing work) writes something similar or even identical, there is no infringement.

Copyright infringement disputes often hinge on whether it is plausible to suppose that someone would come up with an identical program on their own or whether the resemblance between the programs is so close as to be explained only by direct copying. Although this is an extremely difficult hurdle for a copyright holder to overcome in pursuing an infringement action, copyright holders have been successful in overcoming it. Perhaps the most famous case is that of *Franklin v. Apple,* decided by the United States Supreme Court in 1984. Apple was able to show that Franklin copied Apple's operating system (the object code of their operating system) because Franklin's operating

system contained line after line of identical code. Franklin had not even bothered to delete segments of code that included Apple's name.[2]

Franklin v. Apple is also important because it was the decision establishing that computer programs in object code are copyrightable. Before this case was decided, no one was sure whether object code would count as expression since object code cannot be "read" by a human. The Supreme Court decided that machine-readable code counted as "expression."

If a copyright holder goes to court to prove infringement and establishes only that there is a resemblance but not a *striking* resemblance between the copyright holder's software and the defendant's software, then the copyright holder can try to win the case in a different way. The law allows copyrighted material to be appropriated by others without permission if something significant is added to the copyrighted material, such that a new expression is produced. The law makes a distinction between improper and proper appropriation. You are entitled to draw on a copyrighted work as long as you create or add something new. While *improper appropriation* is not a precise notion, it is meant to define the line between taking another's work and building on another's work. Hence, a copyright holder can stop an infringer if the copyright holder can show that the accused relied heavily on the copyright holder's program *and* did not add anything significant to it. This shows improper appropriation.

In either case—whether using striking resemblance or improper appropriation—the copyright holder must also show that the defendant had access to the program. This casts doubt on the possibility that the defendant created the program on his or her own. As already explained, if the defendant produced a similar program on his or her own, then there is no infringement. In the case of computer software, access is often easy to prove, especially if the copyrighted software has been widely marketed. If on the other hand, the software has been kept out of the marketplace, as, for example, when a company produces software for its own internal use and later finds a competitor has identical software, access will not be so easy to prove.

Since the invention of computers and the creation of software, copyright law has been clarified in certain areas, but new issues continue to arise. While there are too many issues to cover here, several can be mentioned to illustrate the diversity and complexity of copyright disputes.

In pursuit of protection for functional software, various aspects of software have come into focus as the valuable part of the software. Both the "structure, sequence, and organization" of programs and the "look and feel" of user interfaces have been seen as potentially what should be protected by copyright. In *Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.*, Whelan had developed a program for Jaslow (for the business activities of his dental office) with the understanding that she (Whelan) would own the rights in the program (*Whelan v. Jaslow,* 1987). A few years after the program had been completed, Jaslow decided

[2] This case established that copyright applies to operating systems programs as well as applications programs.

that there would be a market for a program like the one Whelan had developed for him, but in a different language and for use on a personal computer. Jaslow developed such a program. While he did not use any of the code written by Whelan, he, evidently, studied how Whelan had organized the program. Whelan sued Jaslow for copyright infringement. The court found infringement based on "comprehensive nonliteral similarity." That is, it found that "copyright protection of computer programs may extend beyond a program's literal code to its structure, sequence and organization."

Yet other more recent cases have attempted (and in many cases succeeded) in extending copyright protection to the look and feel of a program and other "nonliteral" elements of programs. These include general flow charts and "more specific organization of intermodular relationships, parameter lists, and macros" as well as program outputs such as "screen displays and user interfaces, menus, and command tree structures contained on screens" (*O. P. Solutions v. Intellectual Property Network Ltd.*, 1999). In addition, endless copyright issues have arisen around the use of images and text on the Web. Needless to say, these issues are made even more complex by the global scope of the Internet.

Copyright continues to be an important legal mechanism for protecting software, though, as you can see, it is far from an ideal form of protection for computer software.

Trade Secrecy Laws

Trade secrecy laws vary from jurisdiction to jurisdiction but in general what they do is give companies the right to keep certain kinds of information secret. The laws are aimed specifically at protecting companies from losing their competitive edge. Thus, for example, a company can keep secret the precise recipe of foods that it sells or the formula of chemicals it uses in certain processes.

Trade secrecy laws were not designed with computer technology in mind and, consequently, their applicability to the computer industry has been somewhat unclear. Nevertheless, this form of protection is used.

To hold up in court, what is claimed as a trade secret typically must (1) have novelty, (2) represent an economic investment to the claimant, (3) have involved some effort in development, and (4) the company must show that it made some effort to keep the information a secret. Software can qualify for such protection; that is, software can be novel, involve effort in development, and require significant investment. Some software companies try to keep their software secret by using nondisclosure clauses in contracts of employment and by means of licensing agreements with those who use their software. Nondisclosure clauses require employees to refrain from revealing secrets that they learn at work. An employee promises not to take copies of programs or reveal the contents of programs owned by the firm, even when he or she leaves the firm. With licensing agreements, companies do not actually sell their software but license its use. Those who want such licenses are required to agree not to do anything that will reveal

the "secret." They agree, that is, not to give away or sell copies of the software that has been licensed.

In addition to employment contracts and licensing agreements, program developers have employed a variety of technical devices to protect their secrets. Such devices include limiting what is available to the user (i.e., not giving the user access to the source program), or building into the program identifying codes so that illegal copies can be traced to their source.

Though this form of protection is used by the software industry, many complain that even with improved technical devices for maintaining secrecy, the protection offered is not adequate. For one thing, the laws are not uniform throughout the United States and this makes it difficult for businesses that operate in multiple jurisdictions. Also, the protection provided is still uncertain because the laws were not designed for computer technology. Thus, companies have to take a risk that the courts will support their claims when and if they are ever tested in the courts.

Most important is the problem of meeting the requirement of maintaining secrecy. Enforcing employment agreements and licensing agreements is a tricky business. Violators can be caught taking or selling direct copies of programs, but there is nothing to stop an employee of one firm from taking the general knowledge and understanding of the principles used in a program to a new job at another firm. Likewise someone who works with licensed software may grasp general principles that can be used to create new software. So while agreements can be made, there are always ambiguities in what one agrees not to reveal.

These problems are not unique to computer software. In a competitive environment, many types of information are better kept secret. What is somewhat unusual about computer software is that often a company must reveal the secret in order to sell the software. That is, in order to provide a licensee with usable software, the software must be modifiable for the licensee's unique needs, and sometimes the only way to do this is to give the licensee access to the source code. The licensee can, then, alter the source code to fit its unique situation. Once the source program is available, the secret is less likely to remain secret, and trade secrecy laws may not apply.

In the typical case described in Scenario 6.2, trade secrecy laws would have been helpful to Bingo, but only to a point. Bingo could have kept the design of its new operating system secret during its development by means of nondisclosure clauses in employment contracts. Once the system was ready for marketing, however, keeping it a secret would have been more difficult. In showing the system to potential users, some information would be revealed, and once the system was in widespread use, Bingo's control of the situation would weaken significantly. Companies that have licenses to use the system cannot police their employees; they cannot entirely control what they see or even what they copy and the general principles used in the system would become known to many users even without malicious intent.

Many companies try to counter this problem by making all modifications to the system for each customer and doing all repairs and maintenance themselves. Provisions for this can be built into the licensing agreement, which minimizes the licensees' exposure to the source code. It works to some extent, though there are still problems in the gray areas of employment agreements and it is not at all practical for small, less complicated programs, where it is impractical for the company to modify every copy sold.

Thus, trade secrecy offers a strong form of protection insofar as it allows the owner to keep software out of the public realm. The problem is that it is often not possible to use trade secrecy because the software has to be put into the public realm in order to be sold or licensed.

Patent Protection

In principle, patent protection offers the strongest form of protection for software because a patent gives the inventor a monopoly on the use of the invention. A patent gives the patent holder both the right to exclude others from making, using or selling the invention, and the right to license others to make, use, or sell it. Even if someone else invents the same thing independently, without any knowledge of the patent holder's invention, the secondary inventor is excluded from use of the patented device without permission of the patent holder. A patent is a legitimate monopoly. Patent protection would seem to give Bingo Software, in Scenario 6.2, the power to prevent Pirate Pete's from marketing its system.

There are three types of patents: utility patents, design patents, and patents on plant forms. The primary concern here is with utility patents for utility patents hold the most promise of protecting software. Utility patents are granted for a term of seventeen years, though the term may be extended for an additional five years.

When it comes to software, the problem with patent protection is not in the kind of protection it provides, but rather the apparent inappropriateness of patents on software. The courts have been reluctant to grant patents on software, and for good reason. To understand the good reasons for not extending patent protection to software, it is helpful to consider the aims and purposes of the patent system.

The aim of the patent system is not simply to ensure that individuals reap rewards for their inventions. Rather, the ruling principle behind the patent system is the advancement of the useful arts and sciences. The objectives of the patent system are to foster invention, to promote disclosure of inventions, and to assure that ideas already in the public domain remain there for free use. The furthering of these objectives is, in turn, expected to improve the economy, increase employment, and generally make better lives for citizens.

One way to encourage invention is to have a system that rewards it. Patent protection does not guarantee that individuals will be rewarded for their

inventions. It provides a form of protection that is a precondition of reward. In other words, if you have a monopoly *and* if your invention has commercial value, then you (and no one else) will be in a position to market the invention. By assuring the possibility of reaping rewards, patent protection encourages invention and innovation. Allowing inventors to profit from their inventions is a means, not an end.

The patent system recognizes that inventions brought into the public realm are beneficial not just in themselves but also because others can learn from and build on these inventions. If new ideas are kept secret, progress in the useful arts and sciences is impeded. Patent protection encourages the inventor to put her ideas into the public realm by promising her protection that she wouldn't have if she simply kept her ideas to herself. If an inventor chooses not to patent his or her invention but to keep it secret, then the inventor will not have recourse if someone else comes up with the same idea or copies the invention. There is nothing to prevent a copier from patenting and marketing the invention.

These two arguments—that patents encourage invention and encourage the bringing of inventions into the public realm—also lead, however, to important restrictions. Abstract ideas, mathematical algorithms, scientific principles, laws of nature, and mental processes cannot be patented. To give someone the exclusive right to use these kinds of things would inhibit further invention rather than fostering it because these things are the building blocks of invention. If individuals had to get permission from a patent holder to use an idea or an algorithm or a law of nature, invention would be significantly impeded. Imagine, for example, the enormous power of the person who held a patent on the law of gravitational pull or the mental steps involved in addition or multiplication!

While this restriction on what can be patented seems more than reasonable, it has caused problems for the patenting of software. A patent claim must satisfy a two-step test before a patent is granted. The claim must (1) fall within the category of permissible subject matter and (2) satisfy three separate tests: (a) it must have utility; (b) it must have novelty; and, (c) it must be non-obvious. The latter three tests are not easy to pass, but they have not been problematic for computer software. That is, many software programs are useful, novel, and not so simple as to be obvious to the average person. Rather, software has had difficulty passing the first step and qualifying as permissible subject matter.

The subject matter of a patent is limited to "a process, machine, manufacture or composition of matter or . . . an improvement thereof." Generally, software has been considered a process or part of a process:

> That a process may be patentable, irrespective of the particular form of the instrumentalities used, cannot be disputed . . . A process is a mode of treatment of certain materials to produce a given result. It is an act, or a series of acts, performed upon the subject matter to be transformed and reduced to a different state or thing. If new and useful, it is just as patentable as is a piece of machinery. (*Cochrane v. Deener*, 94 U.S. 780, 787-788:161876:17)

One difficulty in extending patent protection to software has been in specifying what subject matter is transformed by software: Data? The internal structure of the computer? And so on. However, this problem is secondary to a larger issue.

In the 1970s and 1980s, there was reluctance to grant patents on software or software-related inventions for fear that in granting patents on software, ownership of mental processes might, in effect, be granted. Each of the steps in an algorithm is an operation a human can, in principle at least, perform mentally. If a series of such steps was patented, the patent holder might be able to require that permission or a license be sought before those operations were performed mentally. Needless to say, this would significantly interfere with freedom of thought.

The fear of interfering with mental steps was fairly quickly overcome, but it was followed by concern about patent holders acquiring patent rights to computer and/or mathematical algorithms. Granting a monopoly on the use of a software invention could, it was feared, lead to a monopoly on the use of a mathematical algorithm. This is explicitly prohibited in patent law as inappropriate subject matter. The problem is, what is a software invention if not an algorithm—the order and sequence of steps to achieve a certain result. The issue goes to the heart of what exactly one owns when one has a patent on a piece of software.

Before the *Diamond v. Diehr* case was settled in 1981, very few patents had been granted on computer software (*Diamond v. Diehr*, 1981). There had been a struggle between the United States Supreme Court, the Patent Office, and the Court of Customs and Patent Appeals (CCPA), with the former two resisting granting of patents and the latter pressing to extend patent protection to software. In *Diamond v. Diehr*, the Supreme Court, on only a 5 to 4 vote, denied a patent to Diehr. Even though it was a close and disputable decision, the Patent Office and especially the CCPA interpreted the court's reasoning so as to justify granting patents on software inventions. Although only a handful of software-related patents had been granted before *Diamond v. Diehr*, many thousands have been granted since. Browning (1993) reported that over 9,000 software patents were granted between late-1960s and the end of 1992, and 1,300 were issued in 1992 alone. Statistics vary widely but one recent source reports that "The U.S. Patent and Trademark office bestows more than 20,000 software patents annually" (*Atlanta Constitution*, April 2, 2000).

As mentioned earlier, concerns are now being expressed that too much is being patented and that patents are getting in the way of development in the field. These concerns go to the heart of the patent system's aim, for they suggest that because so much is owned, innovation is now being inhibited. The subject matter limitation on what can be patented aims to insure that the building blocks of science and technology should not be owned so that continued development will flourish, yet complaints suggest just that: the building blocks may now be owned.

The situation can be described roughly as follows: Because so many patents have been granted, before putting new software on the market a software developer must do an extensive and expensive patent search. If overlapping patents are found, licenses must be secured. Even if no overlapping patents are found, there is always the risk of late-issuing patents. Patent searches are not guaranteed to identify all potential infringements because the Patent Office has a poor classification system for software. Hence, there is always the risk of lawsuit due to patent infringement. You may invest a great deal in developing a product, invest more in a patent search, and then find at the last minute that the new product infringes on something already claimed. These factors make software development a risky business and constitute barriers to the development of new software. The costs and risks are barriers especially for small entrepreneurs.

A number of important legal scholars have argued that the Patent Office and the CCPA have overextended the meaning of the Supreme Court's decision in *Diamond v. Diehr* (Samuelson, 1990). The League for Programming Freedom (LPF) (1990) proposes that we pass a law that excludes software from the domain of patents. The situation seems to call for change. Yet, at this point, change may be difficult simply because the computer industry has grown and solidified in an environment structured by these forms of protection.

Summary of Legal Mechanisms

At present, then, there are three forms of legal protection available to software developers, but there are drawbacks to the use of each. Copyright does not give software developers a monopoly on their software and does not protect the valuable part of a program, its functionality. Moreover, copyright law does not provide a stable environment for software developers in that it leaves a good deal of uncertainty about what one owns when one has a copyright on computer software. Trade secrecy (together with employment contracts and licensing agreements) protects all aspects of software by keeping the software out of the public domain. The problem here is that trade secrecy is not always useful to software developers because often they cannot market their software without revealing the secret, and once the secret is out, the law may not apply. Moreover, trade secrecy has the disadvantage that in the long run society does not receive the benefit of exposure to the ideas in the software. Patents promise the strongest form of protection, a monopoly, but the process of acquiring a patent is long, expensive, and fraught with uncertainty. Moreover, extensive use of patent protection may do as much harm as good for software development by interfering with the building blocks of innovation.

This unsatisfactory situation arises because computer software does not fit neatly into the traditional categories employed in property law. Computer software seems to defy the distinction between idea and expression and the distinction between patentable and unpatentable subject matter. Neither copyrights nor patents protect the most valuable aspect of software—the functionality and/or the algorithm. To understand the implications of this situation and look for alternative solutions, it is helpful to think about the philosophical roots of our ideas about property.

THE PHILOSOPHICAL BASIS OF PROPERTY

Property is by no means a simple notion. It is, effectively, created by laws specifying what can and cannot be owned, how things may be acquired and transferred, what owners can and cannot do with their property, and so on. Laws define what counts as property and create different kinds of property. The laws regulating ownership of an automobile, for example, are quite different from those regulating ownership of land. In the case of land, there are rules about how far the land goes down into the ground and how far up into the air space above, about what can and cannot be constructed on the land, when the land may be confiscated by the government, and so on. With automobiles, you may have to show proof of insurance in order to take possession of a car, and even if you own one, you cannot drive it on public roads unless you have a license. Thus, what it means to have property and what it means to own something are rather complex matters.

Software has challenged our traditional notions of property and ownership. American laws, at least, do not grant property rights in the most valuable aspects of software. American laws do not give software developers the kind of protection they claim to need to successfully market innovative software inventions. In debates about whether or not and how software should be protected, assumptions are implicitly made about moral (not just legal) rights in property. These assumptions need to be uncovered, articulated, and critically examined. Drawing on the analysis presented in Chapter 2, on philosophical ethics, two distinct theories of property can be articulated. One theory is consequentialist in justifying the assignment of property rights by the good consequences that result. The other theory is not Kantian though it is closer to Kant in deriving a right to property from a prior, natural right to autonomy and/or personhood. This is the labor theory of property.

It should be clear from the earlier discussion of the legal environment for software ownership that the reasoning behind both the patent and copyright systems is consequentialist. Both systems aim to encourage and facilitate creativity and innovation so that new expressions and new products and processes will be created and made available. Both systems are designed to produce the good consequences of ongoing technological progress. Nevertheless, many discussions of property rights assume that property is not a matter of social utility but a matter of natural right. For this reason, it may be helpful to begin with an examination of the natural rights approach and its implications for software ownership.

Natural Rights Arguments

Drawing on the natural rights tradition, an argument on behalf of ownership of software could be made as follows. A person has a natural right to what he or she produces and this natural right ought to be protected by law. John Locke's labor theory of property fills in this claim. According to Locke, a person acquires a right of ownership in something by mixing his or her labor with it. In a state of nature (before laws and civilized society), an individual who came upon a stretch of land and spent months cultivating the land by planting seed, tending to the plants each day, nourishing them, and protecting them from bad weather would have rights in the crops that grew. The laborer would have a right to the crops because his or her labor produced them. The crops would not have existed without this labor. More to the point, it would be wrong for someone else to come along and seize the crops without permission from the laborer. The thief, in effect, confiscates the laborer's labor.

This Lockean account is intuitively appealing. It appeals to a notion of individual sovereignty and self-ownership. Individuals own themselves or, at least, cannot be owned by another. Since one's labor is an extension of one's body, one's labor cannot be owned by another. The argument goes to the heart of the prohibition on slavery for if an individual puts her labor in something and then someone else takes it, the laborer has been rendered a slave.

Using this Lockean account, it would seem that a software developer could argue that the software he or she develops is rightfully his or her property because it was created from his or her labor. Indeed, the unfairness we sense in Scenario 6.2 derives precisely from the fact that Bingo's labor (and investment of resources acquired by prior labor) has been confiscated by Pirate Pete's. Pirate Pete's has effectively rendered Bingo its slave. This seems a powerful argument for granting some sort of property right in computer software, a property right that would prevent Pirate Pete's from confiscating Bingo's labor.

Critique of the Natural Rights Argument

Yet despite its appeal, the natural rights argument has several flaws especially when it comes to computer software. First, the argument is careless in the way it connects ownership to labor. It would seem that the connection has to be justified. Imagine a society in which individuals do not acquire rights to what they create with their labor. Imagine that no one owns the products of their labor or the products of anyone else's labor. Such a society would not, necessarily, be unjust. Yes, it would be unjust if some individuals acquired rights to what other individuals created, but if there are no property rights whatsoever, then it would seem there is no injustice. Individuals would know that when they mixed their labor with something, they lose their labor. Robert Nozick makes this point, questioning the connection between labor and ownership in his book, *Anarchy, State and Utopia* (1974):

> Why does mixing one's labor with something make one the owner of it? Perhaps because one owns one's labor, and so one comes to own a previously unowned thing that becomes permeated with what one owns. Ownership seeps over into the rest. But why isn't mixing what I own with what I don't own a way of losing what I own rather than a way of gaining what I don't? If I own a can of tomato juice and spill it in the sea so that its molecules (made radioactive, so I can check this) mingle evenly throughout the sea, do I thereby come to own the sea, or have I foolishly dissipated my tomato juice? (Nozick, pp. 174–175)

While Nozick simply asks a question, the question reveals the possibility of a disconnection between labor and property. It shows that there is not a natural connection between your labor and rights to those things created by your labor. A just world in which individuals do not own the products of their labor is plausible.

Nozick's point does not apply in a world in which some property rights are recognized. It does not eliminate the injustice of one person becoming the owner of something created by the labor of another. In other words, there may be nothing unjust about a world in which no one has property rights to anything, but in a world in which there is property, it is wrong for one person to take another's labor. In the latter case, the laborer is rendered a slave. So, Locke's labor theory seems valid in a world of property rights even though we may not need property rights to have a just society. Putting this in terms of Scenario 6.2, if Pirate Pete's copied the software developed by Bingo and then let others copy the software ad infinitum and if there were no laws prohibiting this, there is no natural injustice. However, if Pirate Pete copies Bingo's software, declares the software their property (according to law), and sells the software to others, Pirate Pete's has confiscated Bingo's labor and this seems unfair. So, Locke's labor theory does have implications for the ownership of software.

A second counter to the labor theory directly attacks the confiscation issue. We have to distinguish tangible and nontangible property. Even if the labor theory applies to tangible property (e.g., crops, land, machines), it does not apply to intellectual or nontangible things. In the case of intellectual things such as ideas, musical tunes, and mental steps, more than one person can have or use these things at the same time. If someone comes along and takes or eats the crops that I have grown, I lose the products of my labor altogether. On the other hand, if I labor in creating a song or formalizing an abstract idea such as the Pythagorean theorem and someone hears the song (even memorizes it) or comprehends the idea (and can remember it and use it), I do not lose the song or the theorem. I can continue to have and use these things while others have and use them. So, in the case of intellectual, nontangible creations, the labor expended in creating the thing is not confiscated when another takes the thing. The laborer still has the products of his or her labor.

This is precisely what is at issue with software. Software is intelligible as a nontangible entity. The software developer can describe how the software

works and what it does. Another person can comprehend the idea of the software, even its functionality, and then use this understanding to write original software (source code) that does what was described. Moreover, once a piece of software is developed, many others can make identical copies of the software and yet not deprive the developer of the software.

At the beginning of this chapter and earlier in this book, I mentioned that the reproducibility of computer software has challenged traditional moral and legal notions. In the context of property rights, it should now be clear that this feature of software is both old and new. It is not new in that all forms of intellectual property are reproducible. You and I and many others can all, at the same time, possess a poem written by someone else; that is, we can posses it in the sense of knowing it and being able to recite it. Nontangible things such as ideas, expressions of ideas, mental steps, and music can be possessed and used by many simultaneously, and without interfering with possession by others, including the creator. Yet, while software is not unique in being reproducible, it does have distinct features. Software is a new species of nontangible, reproducible entity. No prior nontangible entity, for example, was capable of transforming the internal structure of an electronic machine into a powerful information-processing device.

Because of the reproducibility of computer software, the labor theory of property cannot be used to justify the assignment of property rights to software developers. If I create a complex piece of software and you copy it and use it, I am *not* deprived of the product of my labor. It is worth noting here that the reproducibility of software is often commended for having this quality in that it offers the potential for broad, democratic distribution.

It is also worth noting that from a natural rights point of view, there seems nothing immoral about a society in which there are no intellectual property rights. Justice prevails if all citizens of such a society know that they cannot own their intellectual creations and can have free access to the intellectual creations of others. Such a society would be open to the criticism that its system does not maximize good consequences, but that is a matter I will take up in a moment.

Even though Locke's labor theory does not provide a justification for property rights in computer software, our analysis of the theory's application to software has cleared the way to the heart of the software ownership issue. Once we concede that software developers do not lose the products of their labor when others copy their software, the core issue becomes clear. While software developers do not lose their software, they do lose something very valuable; they lose *the capacity to sell (and make money from) their creations.* While Bingo in Scenario 6.2 is not deprived of the software it created when Pirate Pete's copies it, Bingo is deprived of the capacity to make money from the software it created. Pirate Pete's is able to sell the product of Bingo's labor (and more) at a lower price. Why would anyone buy something they can acquire at less cost?

What software developers want is not just a right to own (in the sense of possess and use) the products of their labor. They want a right that gives them the capacity to sell and, if successful at selling, make a profit from their creation.[3] This is a claim to an *economic* right, that is a right within an economic system.

Establishing a right to the capacity to sell something requires more than showing that one has labored over the thing. Economic rights are social, not moral or natural rights. Economic rights are created by means of laws that regulate the marketplace. Such laws specify what can and cannot be put into the stream of commerce, under what conditions, meeting what standards, and so on. In the United States, for example, certain drugs are prohibited from being put into the stream of commerce except under very narrowly circumscribed conditions. Similarly, human organs may not be bought and sold in the United States. Another interesting example, counter to the labor theory, is children. Children are, in part at least, the products of their parents' labor, but parents lose all rights to their children when they become adults. Even while they are children, they cannot be sold by their parents. In any case, the point is that a right to sell something is derivative from a complex set of rules structuring commercial activity.

Both the failure of the natural rights argument and the recognition that the right at issue is an economic right, point in the direction of the software ownership issue being best understood in a consequentialist framework. That is, determining whether or not software developers should have a right to sell and profit from their creations is best understood as a matter of consequences, the good and/or bad consequences that will result from granting or not granting such a right. Indeed, the earlier discussion of copyright and patent law provided just such a rationale in that both those systems are aimed at good consequences, namely, progress in the technological arts and sciences. However, before we move to the consequentialist framework, there is one additional natural rights argument worth putting on the table. This is an argument against the ownership of software.

Against Software Ownership

Some of the early legal literature and several early court cases concerning the ownership of software focused on the idea that a patent on a program might violate "the doctrine of mental steps." This doctrine states that a series of mental operations, like addition or subtraction, cannot be owned. Concern was expressed by lawyers that ownership of software might violate this doctrine because computers perform, or at least duplicate, mental steps. It was recognized

[3] I say right to the *capacity* to profit because we couldn't have a right to profit—the creation might not be successful in the marketplace—we can at most have a right to put some thing into the marketplace in a form such that if consumers bought it, it could be profitable.

that in a computer, these operations are performed very quickly so that in a short time, an enormous number of steps are taken. Nevertheless, the operations performed are in principle capable of being performed by a person. If this is so, then ownership of programs could lead to interference with freedom of thought. Those who were granted patents on programs might surreptitiously acquire a monopoly on mental operations. Even though unintended, the fear was that as patent holders used their patents to stop infringements, the effect might be that the patent holder would come to be seen as the owner of the performance of certain mental steps. In other words, down the road of a series of court cases, there might be a case in which performance of mental operations in a person's mind could be ruled an infringement.

Insofar as this fear is accurate, it is a natural rights argument against ownership of computer software. In the language of natural rights, a person has a natural right to freedom of thought. Ownership of software could seriously interfere with freedom of thought. So, ownership of software should not be allowed.

While this argument does not seem to come into play in the legal arguments made today, it is an important potential implication of software ownership to keep in mind. It may well come into play in the future as expert systems and artificial intelligence become more and more sophisticated. That is, property rights in such systems could be contested on grounds that they will interfere with human thinking.

The natural rights arguments for and against ownership of software have been mentioned here because they are so often implicit in discussions of proprietary rights in software. They are also important to keep in mind because they illustrate some of the special fascination with computer and information technology. Many of the tasks now performed by computer and information technology were thought, before the twentieth century, to be the unique province of human beings. Today, a variety of artificial agents are already available and being developed with ever-increasing sophistication. There are good reasons for not allowing the ownership of thought processes, yet machines that think (in some sense of that term) are now a reality. Hence, concerns about the ownership of mental operations should not be dismissed as trivial.

CONSEQUENTIALIST ARGUMENTS

As already indicated, the arguments for ownership of software seem best framed as arguments for a social and economic right to the capacity to sell, and potentially make money from, the development of computer software. Moreover, the arguments for such a right are best framed in terms of consequences. Turning now to the consequentialist framework, the argument for ownership has two parts: bad consequences result from no ownership, and good consequences result from ownership.

Bad Consequences from No Ownership

Those who favor ownership of software are quick to point to the negative effect of no ownership, namely that individuals and companies will not invest their time, energy, and resources to develop and market software. Innovation and development will be impeded, even brought to a standstill. Why develop a new program if the moment you introduce it, others will copy it, produce it more cheaply, and yours will not sell? If we, as a society, want software developed, we will have to give those who develop it the protection they need. Otherwise, society will lose. The great promise of computer and information technology will not be realized. So the argument goes.

This is an important and powerful argument. In short, the argument claims that software will not be developed unless there is an incentive to create it, and it presumes that the only incentive to develop software is to make money. If there is no potential to make money from software development, there will be no software development. Again, so the argument goes.

While the argument is strong, it is important to recognize that it is not quite as powerful or accurate as it may seem. Software development would not come to a complete standstill if there were no ownership because making money is not the only incentive to create software. For one thing, individuals create software because they enjoying creating it and because they need software for various purposes. Once a person creates a piece of software, the person can simply give it to whoever wants it. The creator can make it available on the Web. Such software is sometimes referred to as *freeware*. Freeware is an example of software being created independent of any money-making incentive or any property rights.

Moreover, instead of using making money as the primary incentive for software development, a credit system could be instituted. In a credit system, individuals register their creations and, thereby, claim credit for having created something. If others like what has been created, this adds to the good reputation and pride of the creator. A type of credit system is used in science for scientific discoveries and scientific publications. When individuals publish the results of their research, they are given credit and recognition for their work. In the case of software development, an informal credit system seems already to exist in the sense that many individuals in the field know who developed what and they admire individuals for their accomplishments. Also there are hackers who create software just to show that they can do it or to show that something can be done.

Admittedly, credit may not motivate everyone and it may not create enough of an incentive to promote the development of many expensive and elaborate systems. Still, the point is that making money is not the only incentive for software development and software development would not come to a standstill if software were declared un-ownable.

Indeed, if the focus of attention were more on creating a system that would encourage the development of software, it would seem that private ownership of

software would be only one of many alternatives. *Shareware* is another example of an alternative system. Shareware is software that is initially made available free of charge. Users are encouraged to make copies; shareware can generally be downloaded from the Web. Users are then encouraged to voluntarily pay a small price if they like the software. Sometimes instead of charging for the software itself, the developer will offer to support the software and provide printed documentation to those who register and pay a fee.[4]

In addition to credit and shareware, other possible systems for encouraging software development were seriously considered in the early days of computing. Perhaps the most interesting idea was to make software available free of charge and focus the market on hardware. Computers are useless without software, so hardware companies would be compelled to create good software in order to make their computers marketable. In this environment, there would be an incentive to create software, and the incentive would persist since the better the software available for a type of computer, the better the computer is likely to do in the marketplace.

I am not arguing that any of these alternatives would *necessarily* be better than what we have now. I mention them only to show that software development would not come to a standstill were software to be declared un-ownable. How much and what kind of software would be developed in an environment of no ownership would depend on what alternatives were adopted, if any, to promote software development.

Good Consequences from Ownership

Even though the bad consequences argument is not as strong as it may seem, when it is coupled with the argument pointing to the good consequences from ownership of software, the case is quite persuasive. The argument for the good consequences of ownership has already been given in the discussion of copyright, patent, and trade secrecy. The rationale for creating these forms of legal protection for inventions, expressions, and processes is that giving these protections—these property rights—will encourage invention, innovation, new products, and creative expression. All three systems of law aim at promoting development in the technological arts and sciences. The copyright and patent systems also recognize the value of encouraging inventors and authors to put their creations into the public domain, where others can learn from and build on them, facilitating further invention.

Patent and copyright law have long traditions and there is lots of evidence to suggest that the technological arts and sciences have, in fact, progressed as a result of inventors and authors using these forms of legal protection. Nevertheless, when it comes to software, caution is in order.

[4] Perhaps, the best example of successful shareware is the Linux operating system.

Remember that both the patent and copyright systems limit what can and cannot be owned. Both recognize that the very thing we want to encourage—development in the technological arts and sciences—will be impeded if we fail to limit ownership. To avoid this, copyrights are granted only on the expression of ideas, not the ideas themselves, and patents are granted only on applications of ideas and laws of nature, not the ideas or natural phenomena themselves. It is understood that progress in the technological arts and sciences will be impeded if the building blocks of science and technology are owned. Hence, while the good consequences of ownership make a powerful case for creating and protecting property rights in computer software, the case is qualified. Software should be protected but *not* at the cost of giving away the building blocks of science and technology. We should be careful to maintain an environment that is good for future development.

CONCLUSIONS FROM THE PHILOSOPHICAL ANALYSIS OF PROPERTY

This leaves us with the dilemma facing judges, lawyers, software developers, and policy makers. How can a line be drawn between ownable and un-ownable aspects of software such that software developers can own that which is valuable from a marketplace perspective and yet not own that which will interfere with future development in the field? This is no easy dilemma.

As mentioned earlier, several legal scholars and computer scientists believe that an alternative form of legal ownership should be created just for computer software. While this alternative holds great promise, policy makers are resistant to abandoning traditional forms of property. Copyrights and patents seem to have survived the test of time and appear to have successfully served countries that have adopted these forms of property.

Whether or not a new form of protection should be created for computer software is an important question, but an answer to this question would require a more thorough analysis of current case law than is possible here. The aim here was to provide a broad perspective on the issue. I leave it to readers to more fully explore the current copyright and patent case law to determine whether these legal tools are capable of being interpreted and used in a way that will resolve the dilemma of software ownership. Are they capable, that is, of being interpreted in a way that will give software developers the kind of protection they need while at the same time leaving the building blocks for future development un-ownable?

What is clear from the preceding analysis is that software property rights are best understood in a consequentialist framework (and not a natural rights framework). The consequentialist framework puts the focus on deciding ownership issues in terms of effects on continued creativity and development in the field of software. This framework suggests that the courts will have to

continue to draw a delicate line between what should be ownable and what should not be ownable when it comes to software, along the lines already delineated in patent and copyright law.

IS IT WRONG TO COPY PROPRIETARY SOFTWARE?

Whatever conclusion you draw from the previous discussion, currently there is legal protection for computer software in the United States and many other countries of the world. Software is proprietary; individuals and companies can obtain copyrights and patents on the software they develop or keep software they have developed as a trade secret. Software can be put into the stream of commerce and bought and sold. This means that a person who makes a copy of proprietary software without purchasing the software (or in one way or another obtaining permission from the copyright or patent holder) is breaking the law. A person who makes an illegal copy of software violates the legal right of the patent or copyright holder. The question I now turn to is the question whether such an action is morally wrong. Is it wrong for an individual to make a copy of proprietary software?

First, a clarification: making a back-up copy (for your own protection) of software you have purchased is, generally, not illegal. This is not the type of action that will be examined here.

Second, while the issue to be taken up here is an individual moral issue, the individual at issue could be a collective unit such as a company or agency. From a moral perspective, it makes no difference whether the copier is an individual human being or a collective entity such as a company. Scenario 6.1 depicts Mary copying software that John has purchased. A comparable act occurs when a company buys a software package (and does not obtain a license for multiple copies) and makes multiple copies for use within the company.

Making copies of proprietary software is not uncommon. It seems that many individuals intuitively feel that such behavior is not wrong or not seriously wrong. Indeed, it seems that individuals who would not break any other law will make illegal copies of software. Perhaps, this is not so hard to understand. Making a copy of a piece of software is easy and produces no visible harm. It is not like taking a car or a television where taking possession means depriving the owner of their object. In fact, the illegality of the act of copying is quite subtle. In Scenario 6.1 when Mary copies software purchased by John, there is no question that John owns the software. However, when John bought the software, he only bought the right to possess and use it. He didn't buy the right to make copies for use by others. John owns the software but in a very different way than he owns his car and his bicycle or other tangible possessions.

Consider that the intuitive feeling that copying a piece of software is not harmful can be formulated into an argument for the moral permissibility of software copying. The argument can be made as a two-pronged argument:

Software copying is not wrong because (1) there is nothing intrinsically wrong with copying, and (2) it does no harm.

The first claim, (1), is true in the sense that if there were no laws against copying, the act of copying would not be wrong. Earlier I argued that intellectual property rights are not natural but a matter of social utility, and that argument seems to support this idea that copying is not intrinsically wrong. In a state of nature, the act of copying would have no moral significance. Only when there are laws against it does copying have moral significance.

The second claim, (2), is that the act of copying is not wrong because it does no harm. Here also the claim seems to be true if it refers to the actions of individuals in a state of nature. The act of copying does no harm in the sense that in a state of nature it does no physical harm and it doesn't deprive the possessor of his or her possession.

Nevertheless, both of these arguments have limited application. In societies with laws—laws creating legal rights to certain things, when a person's legal rights are violated, the person is harmed. The person is deprived of something they are legally entitled to—be it life, a vote in an election, possession of a material object, or control of intellectual property. In a society with legal rights to computer software, the act of copying software without permission of the copyright or patent holder, is illegal and harms the owner of the software by depriving the owner of his or her legal right to control use of that software. It deprives the software owner of the right to require payment in exchange for the use of the software. This is not physical harm but it is harm nevertheless. Those who continue to think that this isn't harm should consider the case of Bingo in Scenario 6.2.

So, while the intuitive feeling that software copying is not wrong can be understood, software copying is wrong in a world in which there are property rights in computer software. The argument here is interesting and can be further elaborated.

SOFTWARE COPYING IS IMMORAL BECAUSE IT IS ILLEGAL

The argument just given claims that software copying is immoral *because it is illegal.* There are cases in which the illegality of an act and its immorality are independent. For example, the act of intentionally killing another person would be immoral even if it were not illegal. Software copying is not like this; it is *not* immoral independent of it being illegal. As conceded earlier, there is nothing intrinsically immoral about the act of software copying. Rather the immorality of software copying derives from its illegality. Once a system of property laws has been created, individuals who live under those laws have certain rights, and those who break the laws violate the rights of property holders.

You might be resistant to this line of thinking because you think the laws protecting software ownership are bad laws. Given the discussion in previous

sections, you might be worried that I am putting too much weight on the illegality of software copying. After all, you might argue, the laws creating property rights in computer software are not very good. They don't succeed in protecting software developers very well; they don't take advantage of the potential of software; and they lead to bad consequences when they grant ownership of the building blocks of software development.

The problem with this argument is that it implies that it is permissible to break laws whenever they are bad. While there may be cases in which individuals are justified in breaking a bad law, it overstates the case to claim that it is permissible to break any law one deems not good.

There is a rich philosophical literature on why citizens have an obligation to obey the law and when citizens are justified in breaking the law. The literature recognizes that there are conditions in which individuals are justified in breaking the law, but the conditions are limited. The literature suggests that citizens have what is called a *prima facie* obligation to obey the laws of a relatively just state. Prima facie means "all else being equal" or "unless there are overriding reasons." A prima facie obligation can be overridden by higher order obligations or by special circumstances that justify a different course of action. Higher order obligations will override when, for example, obeying the law will lead to greater harm than disobeying. For example, the law prohibiting automobiles from being driven on the left side of the road (as is the case in many countries) is a good law, but a driver would be justified in breaking this law in order to avoid an accident. On this account of a citizen's obligation to obey the laws of a relatively just state, one has an obligation to obey property laws unless there are overriding reasons for breaking them.

Most cases of software copying do not seem to fall into this category. Most of those who make illegal copies of computer software do so because it is so easy and because they don't want to pay for the software. They don't copy software because an overriding moral reason takes priority over adhering to the law.

You might try to frame software copying as an act of civil disobedience. In the United States and many other countries there is a tradition of recognizing some acts of disobedience to law as morally justified. However, acts of civil disobedience are generally justified on grounds that it would be immoral to obey such laws. Obedience to the law would compel one to act immorally or to support immoral institutions.

It would take us too far afield to explore all the possibilities here but let me suggest some of the obstacles to making the case. To make the case, you would have to show (a) that the system of property rights for software is not just a bad system, but is an unjust system. And you would have to show (b) that adhering to those laws compels you to perform immoral acts or support unjust institutions. Making the case for (a) will not be easy. The critique would have to show either that all copyright and patent laws are unjust *or* that these laws when extended to software are unjust *or* that these laws when interpreted in a certain way (for example, giving Microsoft control of so much of the market) are unjust.

If you can make the case for (a), then (b) will become more plausible. That is, if the system of property rights in computer software is unjust, then it is plausible that adhering to such laws might compel to you act immorally. But this will not be an easy case to make since adhering to the law simply means refraining from copying. In other words, you would have to show that refraining from software copying is an immoral act or supports an immoral institution. This seems somewhat far fetched but several authors have tried to make this sort of argument. That is, several authors have argued that there are some circumstances in which *not* copying is wrong.

Richard Stallman (1995) and Helen Nissenbaum (1994) both point to situations in which a friend is having a great deal of trouble trying to do something with information technology and you have software that will solve the friend's problems. Both make the point that not helping your friend when you know how to help and when you have the means to help seems wrong.

Stallman and Nissenbaum both emphasize that the system of ownership we now have discourages and disables altruism. Neither author seems to recognize the harm done to the copyright or patent holder when the copy is made. In fact, the situation described seems to set up a dilemma in which one can choose one harm over another—violate the right of the software owner or fail to help your friend. This seems to push us back to the idea of overriding circumstances. There probably are some circumstances in which making a copy of proprietary software will be justified to prevent some harm greater than the violation of the software owner's property rights. However, those cases are not the typical case of software copying.

The case for the moral permissibility of software copying would be stronger if the system of software ownership were shown to be unjust or if all property rights were shown to be unjust. Stallman seems to hold the latter view. He seems to believe that all property rights promote selfishness and discourage altruism. He might well be right about this. However, if he is right, why pick on software copying as if it were a special case of justified property rights violation?

I am quite willing to grant that there may be situations in which software copying is justified, namely when some serious harm can only be prevented by making an illegal copy of a piece of proprietary software and using it. In most cases, however, the claims of the software owner would seem to be much stronger than the claims of someone who needs a copy to make their life easier.

In order to fully understand my argument, it will be helpful to use an analogy. Suppose I own a private swimming pool and I make a living by allowing others to use the pool for a fee during certain hours of the day. The pool is closed on certain days and open only for certain hours of other days. You figure out how to break into the pool undetected, and you break in and swim when the pool is closed and I am not around. The act of swimming is not

intrinsically wrong, and swimming in the pool does no visible or physical harm to me, or to anyone else.

Nevertheless, you are using my property without my permission. It would hardly seem a justification for ignoring my property rights if you claimed that you were hot and the swim in my pool made your life more tolerable and less onerous. Your argument would be no more convincing if you pointed out that you were not depriving me of revenues from renting the pool since you swam when the pool was closed. Note the parallel to justifying software copying on grounds that it does no harm, makes the copier's life better, and doesn't deprive the owner of revenue since you wouldn't have bought the software anyway.

Now, the argument would still not be convincing if instead of seeking your own comfort, you sought the comfort of your friend. Suppose, that is, that you had a friend who was suffering greatly from the heat and so you, having the knowledge of how to break into the pool, broke in, in the name of altruism, and allowed your friend to swim while you watched to make sure I didn't appear. In your defense, you argue that it would have been selfish for you not to use your knowledge to help out your friend. Your act was altruistic.

There are circumstances under which your illegal entry into my pool would be justified. For example, if I had given permission to someone to swim in the pool while it was closed to the public and that person, swimming alone, began to drown. You were innocently walking by and saw the person drowning. You broke in and jumped into the pool in order to save the drowning swimmer. Here the circumstances justify your violating my property rights.

There seems no moral difference between breaking into the pool and making a copy of a proprietary piece of software. Both acts violate the legal rights of the owner—legal rights created by reasonably good laws. I grant that these laws prevent others from acting altruistically. I concede that private property in general is individualistic, exclusionary, and even selfish. Nonetheless, it is prima facie wrong to make an illegal copy of proprietary software because to do so is to deprive the owner of their legal right, and this is to harm them.

CONCLUSION

The issues discussed in this chapter are both fascinating and important. Our ideas about property are tied to deeply ingrained notions of rights, fairness, and economic justice. Law and public policy on the ownership of various aspects of computer software structure the environment for software development, so it is important to evaluate these laws to insure the future development of computer and information technology.

The issue of the permissibility of making personal copies of proprietary software is also fascinating and important but for different reasons. Here we are forced to clarify what makes an action right or wrong. We are forced to

come to grips with our moral intuitions and to extend these to entities with unique characteristics.

The thrust of this chapter has been to move discussion of property rights in computer software away from the idea that property rights are given in nature, and towards the idea that we can and should develop property rights that serve us well in the long run.

STUDY QUESTIONS

1. Describe the difference between hardware and software.
2. What are the differences among object programs, source programs, and algorithms?
3. Explain the kind of protection offered by copyright, trade secrecy, and patents. What are the advantages and disadvantages of each for developers of computer software?
4. What is meant by improper appropriation in copyright law?
5. Why is it sometimes difficult for employees to keep company information secret?
6. What is Locke's *labor theory* of property? Why doesn't it necessarily apply to ownership of computer software?
7. What natural rights arguments can be made for and against ownership of software?
8. What are the consequentialist arguments for and against ownership of software?
9. What would happen to software development if software were declared unownable?
10. What arguments support the claim that software copying is immoral? What arguments support the claim that software copying is not immoral?
11. The author argues that software copying is immoral because it is illegal. How does she arrive at this conclusion? Are there limits to the applicability of this argument?

SUGGESTED FURTHER READING

BURK, DAN L., "Transborder Intellectual Property Issues on the Electronic Frontier," *Stanford Law & Policy Review*, vol. 6, no. 1 (1994), pp. 9–16.

DAVIS, RANDALL, PAMELA SAMUELSON, MITCHELL KAPOR, and JEROME REICHMAN, "A New View of Intellectual Property and Software," *Communications of the ACM*, vol. 39, no. 3 (March 1996), pp. 21–30.

U.S. Congress, Office of Technology Assessment, *Finding a Balance: Computer Software, Intellectual Property, and the Challenge of Technological Change*, OTA-TCT-527 (Washington, DC: U.S. Government Printing Office, May 1992).

WEB SITES

For materials on the Web, try the Lexis-Nexis search engine.