3 Economic Perspectives on Open Source

Josh Lerner and Jean Tirole

Introduction

In recent years, there has been a surge of interest in open source software development. Interest in this process, which involves software developers at many different locations and organizations sharing code to develop and refine software programs, has been spurred by three factors:

▪ *The rapid diffusion of open source software.* A number of open source products, such as the Apache web server, dominate product categories. In the personal computer operating system market, International Data Corporation estimates that the open source program Linux has from seven to twenty-one million users worldwide, with a 200 percent annual growth rate. Many observers believe it represents a leading challenger to Microsoft Windows in this important market segment.
▪ *The significant capital investments in open source projects.* Over the past two years, numerous major corporations, including Hewlett-Packard, IBM, and Sun Microsystems, have launched projects to develop and use open source software. Meanwhile, a number of companies specializing in commercializing Linux, such as Red Hat, have completed initial public offerings, and other open source companies such as Cobalt Networks, Collab.Net, Scriptics, and Sendmail have received venture capital financing.
▪ *The new organization structure.* The collaborative nature of open source software development has been hailed in the business and technical press as an important organizational innovation.

To an economist, the behavior of individual programmers and commercial companies engaged in open source processes is startling. Consider these quotations by two leaders of the free software and open source communities:

The idea that the proprietary software social system—the system that says you are not allowed to share or change software—is unsocial, that it is unethical, that it is simply wrong may come as a surprise to some people. But what else can we say about a system based on dividing the public and keeping users helpless? (Stallman 1999a, 54)

The "utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. [Parenthetical comment deleted.] Voluntary cultures that work this way are actually not uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom explicitly recognizes "egoboo" (the enhancement of one's reputation among other fans). (Raymond 2001, 564–565)

It is not initially clear how these claims relate to the traditional view of the innovative process in the economics literature. Why should thousands of top-notch programmers contribute freely to the provision of a public good? Any explanation based on altruism[1] only goes so far. While users in less developed countries undoubtedly benefit from access to free software, many beneficiaries are well-to-do individuals or Fortune 500 companies. Furthermore, altruism has not played a major role in other industries, so it remains to be explained why individuals in the software industry are more altruistic than others.

This chapter seeks to make a preliminary exploration of the economics of open source software. Reflecting the early stage of the field's development, we do not seek to develop new theoretical frameworks or to statistically analyze large samples. Rather, we seek to draw some initial conclusions about the key economic patterns that underlie the open source development of software. (See table 3.1 for the projects we studied.) We find that much can be explained by reference to economic frameworks. We highlight the extent to which labor economics—in particular, the literature on "career concerns"—and industrial organization theory can explain many of the features of open source projects.

At the same time, we acknowledge that aspects of the future of open source development process remain somewhat difficult to predict with "off-the-shelf" economic models. In the final section of this chapter, we highlight a number of puzzles that the movement poses. It is our hope that this chapter will itself have an "open source" nature: that it will stimulate research by other economic researchers as well.

Finally, it is important to acknowledge the relationship with the earlier literature on technological innovation and scientific discovery. The open source development process is somewhat reminiscent of "user-driven innovation" seen in many other industries. Among other examples, Rosenberg's

Table 3.1
The open source programs studied

Program	Apache	Perl	Sendmail
Nature of program	World Wide Web (HTTP) server	System administration and programming language	Internet mail transfer agent
Year of introduction	1994	1987	1979 (predecessor program)
Governing body	Apache Software Foundation	Selected programmers (among the "perl-5-porters") (formerly the Perl Institute)	Sendmail Consortium
Competitors	Internet Information Server (Microsoft); various servers (Netscape)	Java (Sun); Python (open source program); Visual Basic, ActiveX (Microsoft)	Exchange (Microsoft) IMail (Ipswitch); Post.Office (Software.com)
Market penetration	55% (September 1999) (of publicly observable sites only)	Estimated to have one million users	Handles ~80 percent of Internet e-mail traffic
Web site:	http://www.apache.org	http://www.perl.org	http://www.sendmail.com

(1976b) studies of the machine tool industry and von Hippel's (1988) studies of scientific instruments have highlighted the role that sophisticated users can play in accelerating technological progress. In many instances, solutions developed by particular users for individual problems have become more general solutions for wide classes of users. Similarly, user groups have played an important role in stimulating innovation in other settings; certainly, this has been the case since the earliest days in the computer industry (e.g., Caminer et al. 1996).

A second strand of related literature examines the adoption of the scientific institutions ("open science," in Dasgupta and David's (1994) terminology) within for-profit organizations. Henderson and Cockburn (1994) and Gambardella (1995) have highlighted that the explosion of knowledge in biology and biochemistry in the 1970s triggered changes in the management of research and development in major pharmaceutical firms. In particular, a number of firms encouraged researchers to pursue basic research, in addition to the applied projects that typically characterized these organizations. These firms that did so enjoyed substantially higher research and development productivity than their peers, apparently because the research scientists allowed them to more accurately identify promising scientific developments (in other words, their "absorptive capacity" was enhanced) and because the interaction with cutting-edge research made these firms more attractive to top scientists. At the same time, the encouragement of "open science" processes has not been painless. Cockburn, Henderson, and Stern (1999) highlight the extent to which encouraging employees to pursue both basic and applied research led to substantial challenges in designing incentive schemes, because of the very different outputs of each activity and means through which performance is measured.[2]

But as we shall argue, certain aspects of the open source process—especially the extent to which contributors' work is recognized and rewarded—are quite distinct from earlier settings. This study focuses on understanding this contemporaneous phenomenon rather than making a general evaluation of the various cooperative schemes employed over time.

The Nature of Open Source Software

While media attention to the phenomenon of open source software has been only recent, the basic behaviors are much older in origin. There has long been a tradition of sharing and cooperation in software development. But in recent years, both the scale and formalization of the activity have expanded dramatically with the widespread diffusion of the Internet.[3] In the following discussion, we highlight three distinct eras of cooperative software development.

The First Era: The Early 1960s to the Early 1980s

Many of the key aspects of the computer operating systems and the Internet were developed in academic settings such as Berkeley and MIT during the 1960s and 1970s, as well as in central corporate research facilities where researchers had a great deal of autonomy (such as Bell Labs and Xerox's Palo Alto Research Center). In these years, programmers from different organizations commonly shared basic operating code of computer programs—source code.[4]

Many of the cooperative development efforts in the 1970s focused on the development of an operating system that could run on multiple computer platforms. The most successful examples, such as Unix and the C language used for developing Unix applications, were originally developed at AT&T's Bell Laboratories. The software was then installed across institutions, either for free or for a nominal charge. Further innovations were made at many of the sites where the software was installed, and were in turn shared with others. The process of sharing code was greatly accelerated with the diffusion of Usenet, a computer network begun in 1979 to link together the Unix programming community. As the number of sites grew rapidly, the ability of programmers in university and corporate settings to rapidly share technologies was considerably enhanced.

These cooperative software development projects were undertaken on a highly informal basis. Typically no effort to delineate property rights or to restrict reuse of the software were made. This informality proved to be problematic in the early 1980s, when AT&T began enforcing its (purported) intellectual property rights related to Unix.

The Second Era: The Early 1980s to the Early 1990s

In response to these threats of litigation, the first efforts to formalize the ground rules behind the cooperative software development process emerged. This movement ushered in the second era of cooperative software development. The critical institution during this period was the Free Software Foundation, begun by Richard Stallman of the MIT Artificial Intelligence Laboratory in 1983. The foundation sought to develop and disseminate a wide variety of software without cost.

One important innovation introduced by the Free Software Foundation was a formal licensing procedure that aimed to preclude the assertion of patent rights concerning cooperatively developed software (as many believed that AT&T had done in the case of Unix). In exchange for being able to modify and distribute the GNU (a "recursive acronym" standing for "GNU's not Unix"), software, software developers had to agree to make the source code freely available (or at a nominal cost). As part of the General Public License (GPL, also known as "copylefting"), the user also had to agree not to impose licensing restrictions on others. Furthermore, all enhancements to the code—and even code that intermingled the cooperatively developed software with separately created software—had to be licensed on the same terms. It is these contractual terms that distinguish open source software from shareware (where the binary files but not the underlying source code are made freely available, possibly for a trial period only) and public-domain software (where no restrictions are placed on subsequent users of the source code).[5]

This project, as well as contemporaneous efforts, also developed a number of important organizational features. In particular, these projects employed a model where contributions from many developers were accepted (and frequently publicly disseminated or posted). The official version of the program, however, was managed or controlled by a smaller subset of individuals closely involved with the project, or in some cases, by an individual leader. In some cases, the project's founder (or a designated successor) served as the leader; in others, leadership rotated between various key contributors.

The Third Era: The Early 1990s to Today

The widespread expansion of Internet access in the early 1990s led to a dramatic acceleration of open source activity. The volume of contributions and diversity of contributors expanded sharply, and numerous new open source projects emerged, most notably Linux (an operating system developed by Linus Torvalds in 1991). As discussed in detail next, interactions between commercial companies and the open source community also became commonplace in the 1990s.

Another innovation during this period was the proliferation of alternative approaches to licensing cooperatively developed software. During the 1980s, the GPL was the dominant licensing arrangement for cooperatively developed software. This situation changed considerably during the 1990s. In particular, Debian, an organization set up to disseminate Linux, developed the "Debian Free Software Guidelines" in 1995. These guidelines allowed licensees greater flexibility in using the program, including the right to bundle the cooperatively developed software with proprietary code. These provisions were adopted in early 1997 by a number of individuals involved in cooperative software development, and were subsequently dubbed the "Open Source Definition." As the authors explained:

> **License Must Not Contaminate Other Software**
>
> The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software. Rationale: Distributors of open-source software have the right to make their own choices about their own software (Open Source Initiative 1999).

These new guidelines did not require open source projects to be "viral": they need not "infect" all code that was compiled with the software with the requirement that it be covered under the license agreement as well. At the same time, they also accommodated more restrictive licenses, such as the GPL.

The past few years have seen unprecedented growth of open source software. At the same time, the movement has faced a number of challenges. We highlight two of these here: the "forking" of projects (the development of competing variations) and the development of products for high-end users.

The first of these two issues has emerged in a number of open source projects: the potential for programs to splinter into a number of variants. In some cases, passionate disputes over product design have led to such splintering of open source projects. Examples of such splintering occurred with the Berkeley Unix program and Sendmail during the late 1980s.

Another challenge has been the apparently lesser emphasis on documentation and support, user interfaces,[6] and backward compatibility in at least some open source projects. The relative technological features of software developed in open source and traditional environments are a matter of passionate discussion. Some members of the community believe that this production method dominates traditional software development in all respects. But many open source advocates argue that open source software tends to be geared to the more sophisticated users.[7] This point is made colorfully by one open source developer:

> [I]n every release cycle Microsoft always listens to its *most ignorant customers*. This is the key to dumbing down each release cycle of software for further assaulting the non-personal-computing population. Linux and OS/2 developers, on the other hand, tend to listen to their *smartest* customers. . . . The good that Microsoft does in bringing computers to non-users is outdone by the curse that they bring on experienced users (Nadeau 1999).

Certainly, the greatest diffusion of open source projects appears to be in settings where the end users are sophisticated, such as the Apache server installed by systems administrators. In these cases, users are apparently more willing to tolerate the lack of detailed documentation or easy-to-understand user interfaces in exchange for the cost savings and the permission to modify the source code themselves. In several projects, such as Sendmail, project administrators chose to abandon backward compatibility in the interests of preserving program simplicity.[8] One of the rationales for this decision was that administrators using the Sendmail system were responsive to announcements that these changes would be taking place, and rapidly upgraded their systems. In a number of commercial software projects, it has been noted, these types of rapid responses are not as common. Once again, this reflects the greater sophistication and awareness of the users of open source software.

The debate about the ability of open source software to accommodate high-end users' needs has direct implications for the choice of license. The recent popularity of more liberal licenses and the concomitant decline of the GNU license are related to the rise in the "pragmatist" influence. These individuals believe that allowing proprietary code and for-profit activities in segments that would otherwise be poorly served by the open source community will provide the movement with its best chance for success.

Who Contributes?

Computer system administrators, database administrators, computer programmers, and other computer scientists and engineers occupied about 2.1 million jobs in the United States in 1998. (Unless otherwise noted, the information in this paragraph is from U.S. Department of Labor 2000.) A large number of these workers—estimated at between five and ten percent—are either self-employed or retained on a project-by-project basis by employers. Computer-related positions are projected by the federal government to be among the fastest-growing professions in the next decade.

The distribution of contributors to open source projects appears to be quite skewed. This is highlighted by an analysis of 25 million lines of open source code, constituting 3,149 distinct projects (Ghosh and Ved Prakash 2000). The distribution of contributions is shown in figure 3.1. More than three-quarters of the nearly 13,000 contributors made only one contribution; only one in twenty-five had more than five contributions. Yet the top decile of contributors accounted for fully 72 percent of the code contributed to the open source projects, and the top two deciles for 81 percent (see figure 3.2). This distribution would be even more skewed if those who simply reported errors, or "bugs," were considered: for every individual who contributes code, five will simply report errors (Valloppillil, 1998). To what extent this distribution is unique to open source software is unclear: the same skewness of output is also observed among programmers employed in commercial software development facilities (e.g., see Brooks 1995 and Cusumano 1991), but it is unclear whether these distributions are similar in their properties.

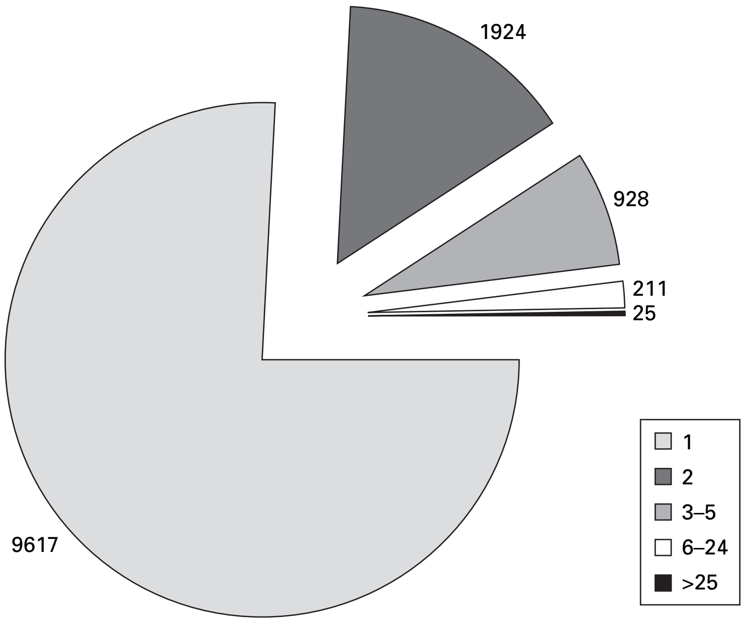
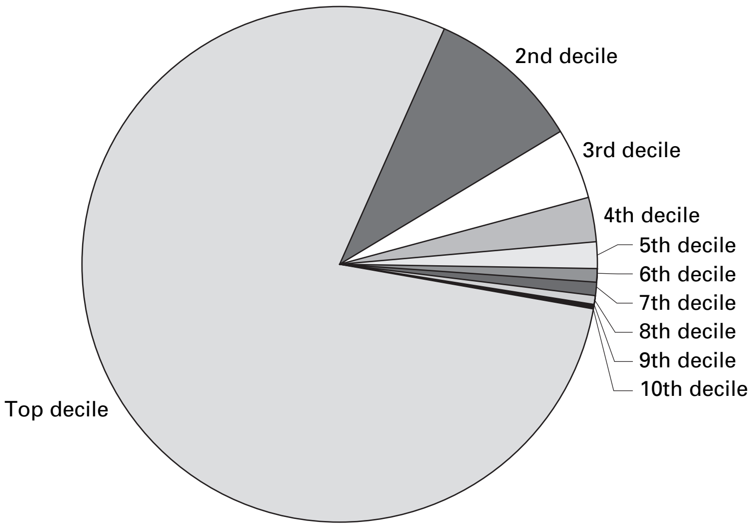


Figure 3.1
Distribution of contributions by participant (Ghosh and Prakash 2000)

The overall picture that we drew from our interviews and from the responses we received in reaction to the first draft of the paper is that the open source process is quite elitist. Important contributors are few and ascend to the "core group" status, the ultimate recognition by one's peers. The elitist view is also supported by Mockus, Fielding, and Herbsleb's (2000) study of contributions to Apache. For Apache, the (core) "developers mailing list" is considered as the key list of problems to be solved, while other lists play a smaller role. The top 15 developers contribute 83 percent to 91 percent of changes (problem reports by way of contrast offer a much less elitist pattern).

Some evidence consistent with the suggestion that contributions to open source projects are being driven by signaling concerns can be found in the analysis of contributors to a long-standing archive of Linux postings maintained at the University of North Carolina by Dempsey et al. 1999. These authors examine the suffix of the contributors' e-mail addresses. While the location of many contributors cannot be precisely identified (for instance, contributors at ".com" entities may be located anywhere in the world), the



Does not include 9% of code, where contrbutor could not be identified.

Figure 3.2
Distribution of code contributed by decile: Does not include 9 percent of code where contributor could not be identified. (Ghosh and Prakash 2000)

results are nonetheless suggestive. As figure 3.3 depicts, 12 percent of the contributors are from entities with an ".edu" suffix (typically, U.S. educational institutions), 7 percent from ".org" domains (traditionally reserved from U.S. nonprofits), 37 percent are from Europe (with suffixes such as ".de" and ".uk"), and 11 percent have other suffixes, many of which represent other foreign countries. This suggests that many of the contributions are coming from individuals outside the major software centers.

What Does Economic Theory Tell Us about Open Source?

This section and the next use economic theory to shed light on three key questions: Why do people participate?[9] Why are there open source projects in the first place? And how do commercial vendors react to the open source movement?

What Motivates Programmers?

A programmer participates in a project, whether commercial or open source, only if he or she derives a net benefit from engaging in the activ-

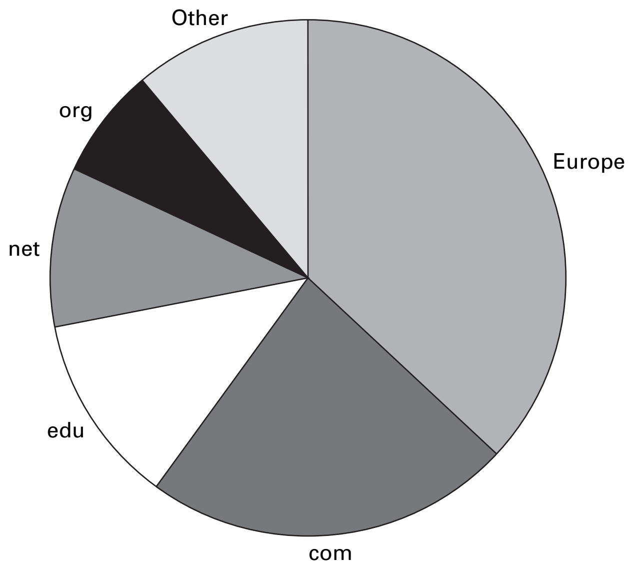


Figure 3.3
Suffix of Linux contributors' e-mail (Dempsey et al. 1999)

ity. The net benefit is equal to the immediate payoff (current benefit minus current cost) plus the delayed payoff (delayed benefit minus delayed cost).

A programmer working on an open source software development project incurs a variety of benefits and costs. The programmer incurs an opportunity cost of time. While is working on this project, a programmer is unable to engage in another programming activity. This opportunity cost exists at the extensive and intensive margins. First, programmers who would work as independents on open source projects would forgo the monetary compensation they would receive working for a commercial firm or a university. Second, and more to the point, for a programmer with an affiliation with a commercial company, a university or research lab, the opportunity cost is the cost of not focusing on the primary mission. For example, the academic's research output might sag, and the student's progress towards a degree slow down; these examples typify delayed costs. The size of this opportunity cost of not focusing on the primary mission of course depends on the extent of monitoring by the employer and, more generally, the pressure on the job.

Two immediate benefits might counter this cost. First, programmers, when fixing a bug or customizing an open source program, might actually improve rather than reduce their performance in the mission endowed upon them by their employer. This is particularly relevant for system administrators looking for specific solutions for their company. Second, the programmer compares the enjoyability of the mission set by the employer and the open source alternative. A "cool" open source project might be more fun than a routine task.

The delayed reward covers two distinct, although hard-to-distinguish, incentives. The *career concern incentive* refers to future job offers, shares in commercial open source-based companies,[10] or future access to the venture capital market.[11] The *ego gratification incentive* stems from a desire for peer recognition. Probably most programmers respond to both incentives. There are some differences between the two. The programmer mainly preoccupied by peer recognition may shun future monetary rewards, and may also want to signal his or her talent to a slightly different audience than a programmer motivated by career concerns. From an economic perspective, however, the incentives are similar in most respects. We group the career concern incentive and the ego gratification incentive under a single heading: the *signaling incentive*.

Economic theory (e.g., Holmström 1999) suggests that this signaling incentive is stronger,

1. the more visible the performance to the relevant audience (peers, labor market, venture capital community)
2. the higher the impact of effort on performance
3. the more informative the performance about talent

The first condition gives rise to what economists call "strategic complementarities." To have an "audience," programmers will want to work on software projects that will attract a large number of other programmers. This suggests the possibility of multiple equilibria. The same project might attract few programmers because programmers expect that others will not be interested; or it may flourish as programmers (rationally) have faith in the project.

The same point applies to forking in a given open source project. Open source processes are in this respect quite similar to academic research. The latter is well known to exhibit fads: see the many historical examples of simultaneous discoveries discussed by Merton (1973). Fields are completely neglected for years, while others with apparently no superior intrinsic interest attract large numbers of researchers. Fads in academia are frowned upon for their inefficient impact on the allocation of research. It should not be ignored, however, that fads also have benefits. A fad can create a strong signaling incentive: researchers working in a popular area may be highly motivated to produce a high-quality work, since they can be confident that a large audience will examine their work.[12]

Turning to the leadership more specifically, it might still be a puzzle that the leader initially releases valuable code to the community.[13] Despite the substantial status and career benefits of being a leader of an important open source project, it would seem that most would not resist the large monetary gains from taking a promising technology private. We can only conjecture as to why this is not the case. One possibility is that taking the technology private might meet layers of resistance within the leader's corporation. To the extent that the innovation was made while working in-house, the programmer must secure a license from the employer;[14] and the company, which does not want to lose a key programmer, might not be supportive of the request. Another possibility is that the open source process may be a more credible way of harnessing energies when fighting against a dominant player in the industry.

Comparison between Open Source and Closed Source Programming Incentives

To compare programmers' incentives in the open source and proprietary settings, we need to examine how the fundamental features of the two environments shape the incentives just reviewed. We first consider the relative short-term rewards, and then turn to the deferred compensation.

Commercial projects have an edge on the current compensation dimension because the proprietary nature of the code generates income, which makes it worthwhile for private companies to offer salaries.[15] This contention is the old argument in economics that the prospect of profit encourages investment, which is used, for instance, to justify the awarding of patents to encourage invention.

By way of contrast, an open source project might well lower the cost for the programmer, for two reasons:

1. *"Alumni effect":* Because the code is freely available to all, it can be used in schools and universities for learning purposes; so it is already familiar to programmers. This reduces their cost of programming for Unix, for example.[16]
2. *Customization and bug-fixing benefits:* The cost of contributing to an open source project can be offset if the activity brings about a private benefit (bug fixing, customization) for the programmer and his or her firm. Note again that this factor of cost reduction is directly linked to the openness of the source code.[17]

Let us now turn to the delayed reward (signaling incentive) component. In this respect too, the open source process has some benefits over the closed source approach. As we noted, signaling incentives are stronger, the more visible the performance and the more attributable the performance to a given individual. Signaling incentives therefore may be stronger in the open source mode for three reasons:

1. *Better performance measurement:* Outsiders can observe only inexactly the functionality and/or quality of individual elements of a typical commercially developed program, as they are unable to observe the proprietary source code. By way of contrast, in an open source project, the outsiders are able to see not only what the contribution of each individual was and whether that component "worked," but also whether the task was hard, whether the problem was addressed in a clever way, whether the code can be useful for other programming tasks in the future, and so forth.
2. *Full initiative:* The open source programmer is his or her own boss and takes full responsibility for the success of a subproject. In a hierarchical commercial firm, though, the programmer's performance depends on a supervisor's interference, advice, and so on. Economic theory predicts that the programmer's performance is more precisely measured in the former case.[18]
3. *Greater fluidity:* It may be argued that the labor market is more fluid in an open source environment. Programmers are likely to have less idiosyncratic, or firm-specific, human capital that limits shifting one's efforts to a new program or work environment. (Since many elements of the source code are shared across open source projects, more of the knowledge they have accumulated can be transferred to the new environment.)

These theoretical arguments also provide insights as to *who* is more likely to contribute and *what tasks* are best suited to open source projects. Sophisticated users derive direct benefits when they customize or fix a bug in open source software.[19] A second category of potential contributors consists of individuals with strong signaling incentives; these contributors might use open source software as a port of entry. For instance, open source processes may give a talented system administrator at a small academic institution (who is also a user!) a unique opportunity to signal talent to peers, prospective employers, and the venture capital community.[20]

As to the tasks that may appeal to the open source community, one would expect that tasks such as those related to the operating systems and programming languages, whose natural audience is the community of programmers, would give rise to strong signaling incentives. (For instance, the use of Perl is largely restricted to system administrators.) By way of contrast, tasks aiming at helping the much less sophisticated end user—design of easy-to-use interfaces, technical support, and ensuring backward compatibility—usually provide lower signaling incentives.[21]

Evidence on Individual Incentives

A considerable amount of evidence is consistent with an economic perspective. First, user benefits are key to a number of open source projects. One of the origins of the free software movement was Richard Stallman's inability to improve a printer program because Xerox refused to release the source code. Many open source project founders were motivated by information technology problems that they had encountered in their day-to-day work. For instance, in the case of Apache, the initial set of contributors was almost entirely system administrators who were struggling with the same types of problems as Brian Behlendorf. In each case, the initial release was "runnable and testable": it provided a potential, if imperfect, solution to a problem that was vexing data processing professionals.

Second, it is clear that giving credit to authors is essential in the open source movement. This principle is included as part of the nine key requirements in the "Open Source Definition" (Open Source Initiative 1999). This point is also emphasized by Raymond 2001, who points out "surreptitiously filing someone's name off a project is, in cultural context, one of the ultimate crimes."

More generally, the reputational benefits that accrue from successful contributions to open source projects appear to have real effects on the developers. This is acknowledged within the open source community itself. For instance, according to Raymond 2001, the primary benefits that accrue to successful contributors of open source projects are "good reputation among one's peers, attention and cooperation from others . . . [and] higher status [in the] . . . exchange economy." Thus, while some of benefits conferred from participation in open source projects may be less concrete in nature, there also appear be quite tangible—if delayed—rewards.

The Apache project provides a good illustration of these observations. The project makes a point of recognizing all contributors on its web site—even those who simply identify a problem without proposing a solution. Similarly, the organization highlights its most committed contributors, who have the ultimate control over the project's evolution. Moreover, it appears that many of the skilled Apache programmers have benefited materially from their association with the organization. Numerous contributors have been hired into Apache development groups within companies such

Table 3.2
Commercial roles played by selected individuals active in open source movement

Individual	Role and company
Eric Allman	Chief Technical Officer, Sendmail, Inc. (support for open source software product)
Brian Behlendorf	Founder, President, and Chief Technical Officer, Collab.Net (management of open source projects)
Keith Bostic	Founder and President, Sleepycat Software
L. Peter Deutsch	Founder, Aladdin Enterprises (support for open source software product)
William Joy	Founder and Chief Scientist, Sun Microsystems (workstation and software manufacturer)
Michael Tiemann	Founder, Cygnus Solutions (open source support)
Linus Torvalds	Employee, Transmeta Corporation (chip design company)
Paul Vixie	President, Vixie Enterprises (engineering and consulting services)
Larry Wall	Employee, O'Reilly Media (software documentation publisher)

as IBM, become involved in process-oriented companies such as Collab.Net that seek to make open source projects more feasible (see following discussion), or else moved into other Internet tools companies in ways that were facilitated by their expertise and relationships built up during their involvement in the open source movement. Meanwhile, many of the new contributors are already employed by corporations and working on Apache development as part of their regular assignments.

There is also substantial evidence that open source work may be a good stepping stone for securing access to venture capital. For example, the founders of Sun, Netscape, and Red Hat had signaled their talent in the open source world. In table 3.2, we summarize some of the subsequent commercial roles played by individuals active in the open source movement.

Organization and Governance

Favorable characteristics for open source production are (a) its modularity (the overall project is divided into much smaller and well-defined tasks ("modules") that individuals can tackle independently from other tasks) and (b) the existence of fun challenges to pursue.[22] A successful open source project also requires a credible leader or leadership, and an organization consistent with the nature of the process. Although the leader is often at the origin a user who attempts to solve a particular program, the leader over time performs less and less programming. The leader must provide a "vision," attract other programmers, and, last but not least, "keep the project together" (prevent it from forking or being abandoned).

Initial Characteristics The success of an open source project is dependent on the ability to break the project into distinct components. Without parcelling out work in different areas to programming teams who need little contact with one another, the effort is likely to be unmanageable. Some observers argue that the underlying Unix architecture lent itself well to the ability to break development tasks into distinct components. It may be that as new open source projects move beyond their Unix origins and encounter new programming challenges, the ability to break projects into distinct units will be less possible. But recent developments in computer science and programming languages (for example the development of object-oriented programming) have encouraged further modularization, and may facilitate future open source projects.

The initial leader must also assemble a critical mass of code to which the programming community can react. Enough work must be done to show that the project is possible and has merit. At the same time, to attract additional programmers, it may be important that the leader does not perform too much of the job on his own and leaves challenging programming problems to others.[23] Indeed, programmers will initially be reluctant to join a project unless they identify an exciting challenge. Another reason why programmers are easier to attract at an early stage is that, if successful, the project will keep attracting a large number of programmers in the future, making early contributions very visible.

Consistent with this argument, it is interesting to note that each of the four cases described previously appeared to pose challenging programming problems.[24] At the initial release of each of these open source programs, considerable programming problems were unresolved. The promise that the project was not near a "dead end," but rather would continue to attract ongoing participation from programmers in the years to come, appears to be an important aspect of its appeal.

In this respect, Linux is perhaps the quintessential example. The initial Linux operating system was quite minimal, on the order of a few tens of thousands of lines of code. In Torvalds' initial postings, in which he sought to generate interest in Linux, he explicitly highlighted the extent to which the version would require creative programming in order to achieve full functionality. Similarly, Larry Wall attributes the much of the success of Perl to the fact that it "put the focus on the creativity of the programmer." Because it has a very limited number of rules, the program has evolved in a variety of directions that were largely unanticipated when Wall initiated the project.

Leadership

Another important determinant of project success appears to be the nature of its leadership. In some respects, the governance structures of open source projects are quite different. In a number of instances, including Linux, there is an undisputed leader. While certain aspects are delegated to others, a strong centralization of authority characterizes these projects. In other cases, such as Apache, a committee resolves the disputes by voting or a consensus process.

At the same time, leaders of open source projects share some common features. Most leaders are the programmers who developed the initial code for the project (or made another important contribution early in the project's development). While many make fewer programming contributions, having moved on to broader project management tasks, the individuals that we talked to believed that the initial experience was important in establishing credibility to manage the project. The splintering of the Berkeley-derived Unix development programs has been attributed in part to the absence of a single credible leader.

But what does the leadership of an open source project do? It might appear at first sight that the unconstrained, quasi-anarchistic nature of the open source process leaves little scope for a leadership. This perception is incorrect. While the leader has no "formal authority" (is unable to instruct anyone to do anything), he or she has substantial "real authority" in successful open source projects.[25] That is, a leader's "recommendations," broadly viewed, tend to be followed by the vast majority of programmers working on the project. These recommendations include the initial "vision" (agenda for work, milestones), the subsequent updating of goals as the project evolves, the appointment of key leaders, the cajoling of programmers so as to avoid attrition or forking, and the overall assessment of what has been and should be achieved. (Even though participants are free to take the project where they want as long as they release the modified code, acceptance by the leadership of a modification or addition provides some certification as to its quality and its integration/compatibility with the overall project. The certification of quality is quite crucial to the open source project, because the absence of liability raises concerns among users that are stronger than for commercial software, for which the vendor is liable).

The key to a successful leadership is the programmers' trust in the leadership: that is, they must believe that the leader's objectives are sufficiently congruent with theirs and not polluted by ego-driven, commercial, or political biases. In the end, the leader's recommendations are only meant to convey information to the community of participants. The recommendations receive support from the community only if they are likely to benefit the programmers; that is, only if the leadership's goals are believed to be aligned with the programmers' interests.

For instance, the leadership must be willing to accept meritorious improvements, even though they might not fit within the leader's original blueprint. Trust in the leadership is also key to the prevention of forking. While there are natural forces against forking (the loss of economies of scale due to the creation of smaller communities, the hesitations of programmers in complementary segments to port to multiple versions, and the stigma attached to the existence of a conflict), other factors may encourage forking. User-developers may have conflicting interests as to the evolution of the technology. Ego (signaling) concerns may also prevent a faction from admitting that another approach is more promising, or simply from accepting that it may socially be preferable to have one group join the other's efforts, even if no clear winner has emerged. The presence of a charismatic (trusted) leader is likely to substantially reduce the probability of forking in two ways. First, indecisive programmers are likely to rally behind the leader's preferred alternative. Second, the dissenting faction might not have an obvious leader of its own.

A good leader should also clearly communicate its goals and evaluation procedures. Indeed, the open source organizations go to considerable efforts to make the nature of their decision making process transparent: the process by which the operating committee reviews new software proposals is frequently posted and all postings archived. For instance, on the Apache web site, it is explained how proposed changes to the program are reviewed by the program's governing body, whose membership is largely based on contributions to the project. (Any significant change requires at least three "yes" votes—and no vetoes—by these key decision-makers.)

Commercial Software Companies' Reactions to the Open Source Movement

This section examines the interface between open and closed source software development. Challenged by the successes of the open source movement, the commercial software corporations may employ one of two strategies. The first is to emulate some incentive features of open source processes in a distinctively closed source environment. Another is to try to mix open and closed source processes to get the best of both worlds.

Why Don't Corporations Duplicate the Open Source Incentives?

As we already noted, owners of proprietary code are not able to enjoy the benefits of getting free programming training in schools and universities (the alumni effect); nor can they easily allow users to modify their code and customize it without jeopardizing intellectual property rights.

Similarly, and for the reasons developed earlier, commercial companies will never be able to fully duplicate the visibility of performance reached in the open source world. At most, they can duplicate to some extent some of the signaling incentives of the open source world. Indeed, a number of commercial software companies (for example, video game companies, and Qualcomm, creators of the Eudora email program) list people who have developed the software. It is an interesting question why others do not. To be certain, commercial companies do not like their key employees to become highly visible, lest they be hired away by competitors.[26] But, to a large extent, firms also realize that this very visibility enables them to attract talented individuals and provides a powerful incentive to existing employees.[27]

To be certain, team leaders in commercial software build reputations and get identified with proprietary software just as they can on open source projects; but the ability of reputations to spread beyond the leaders is more limited, due to the nonverifiability of claims about who did what.[28]

Another area in which software companies might try to emulate open source development is the promotion of widespread code sharing within the company. This may enable them to reduce code duplication and to broaden a programmer's audience. Interestingly, existing organizational forms may preclude the adoption of open source systems within commercial software firms. An internal Microsoft document on open source (Valloppillil 1998) describes a number of pressures that limit the implementation of features of open source development within Microsoft. Most importantly, each software development group appears to be largely autonomous. Software routines developed by one group are not shared with others. In some instances, the groups seek to avoid being broken up by not documenting a large number of program features. These organizational attributes, the document suggests, lead to very complex and interdependent programs that do not lend themselves to development in a "compartmentalized" manner nor to widespread sharing of source code.[29]

The Commercial Software Companies' Open Source Strategies

As should be expected, many commercial companies have undertaken strategies (discussed in this section) to capitalize on the open source movement. In a nutshell, they expect to benefit from their expertise in some segment whose demand is boosted by the success of a complementary open source program. While improvements in the open source software are not appropriable, commercial companies can benefit indirectly in a complementary proprietary segment.[30]

Living Symbiotically Off an Open Source Project One such strategy is straightforward. It consists of commercially providing complementary services and products that are not supplied efficiently by the open source community. Red Hat for example, exemplifies this "reactive" strategy.[31]

In principle, a "reactive" commercial company may want to encourage and subsidize the open source movement; for example, by allocating a few programmers to the open source project.[32] Red Hat will make more money on support if Linux is successful. Similarly, if logic semiconductors and operating systems for personal computers are complements, one can show by a revealed preference argument that Intel's profits will increase if Linux (which, unlike Windows, is free) takes over the PC operating system market. Sun may benefit if Microsofts' position is weakened; Oracle might wish to port its database products to a Linux environment in order to lessen its dependence on Sun's Solaris operating system, and so forth. Because firms do not capture all the benefits of the investments, though, the free-rider problem often discussed in the economics of innovation should apply here as well. Subsidies by commercial companies for open source projects should remain limited unless the potential beneficiaries succeed in organizing a consortium (which will limit the free-riding problem).

Code Release A second strategy is to take a more proactive role in the development of open source software. Companies can release existing proprietary code and create some governance structure for the resulting open source process. For example, Hewlett-Packard recently released its Spectrum Object Model linker to the open source community in order to help the Linux community port Linux to Hewlett-Packard's RISC architecture.[33] This is similar to the strategy of giving away the razor (the released code) to sell more razor blades (the related consulting services that HP will provide).

When can it be advantageous for a commercial company to release proprietary code under an open source license? The first situation is, as we have noted, when the company expects to thereby boost its profit on a complementary segment. A second is when the increase in profit in the proprietary complementary segment offsets any profit that would have been made in the primary segment, had it not been converted to open source. Thus, the temptation to go open source is particularly strong when the company is too small to compete commercially in the primary segment or when it is lagging behind the leader and about to become extinct in that segment.[34,35]

Various efforts by corporations selling proprietary software products to develop additional products through an open source approach have been undertaken. One of the most visible of these efforts was Netscape's 1998 decision to make Mozilla, a portion of its browser source code, freely available. This effort encountered severe difficulties in its first year, receiving only approximately two dozen postings by outside developers. Much of the problems appeared to stem from the insufficiently "modular" nature of the software: as a reflection of its origins as a proprietary commercial product, the different portions of the program were highly interdependent and interwoven. Netscape eventually realized it needed to undertake a major restructuring of the program, in order to enhance the ability of open source programmers to contribute to individual sections. It is also likely that Netscape raised some suspicions by not initially adopting the right governance structure. Leadership by a commercial entity may not internalize enough of the objectives of the open source community. In particular, a corporation may not be able to credibly commit to keeping all source code in the public domain and to adequately highlighting important contributions.[36]

For instance, in the Mozilla project, Netscape's unwillingness to make large amounts of browser code public was seen as an indication of its questionable commitment to the open source process. In addition, Netscape's initial insistence on the licensing terms that allowed the corporation to relicense the software developed in the open source project on a proprietary basis was viewed as problematic (Hamerly, Paquin, and Walton 1999).

(The argument is here the mirror image of the standard argument in industrial economics that a firm may want to license its technology to several licensees in order to commit not to expropriate producers of complementary goods and services in the future: see Shepard (1987) and Farrell and Gallini (1988).) Netscape initially proposed the "Netscape Public License," a cousin to the BSD license that allowed Netscape to take pieces of the open source code and turn them back into a proprietary project again. The licensing terms, though, may not have been the hindering factor, since the terms of the final license are even stricter than those of the GPL. Under this new license (the "Mozilla Public License"), Netscape cannot relicense the modifications to the code.

Intermediaries Hewlett-Packard's management of the open source process seems consistent with Dessein (1999). Dessein shows that a principal with formal control rights over an agent's activity in general gains by delegating his control rights to an intermediary with preferences or incentives that fall between or combine between his and the agent's. The partial alignment of the intermediary's preferences with the agent's fosters trust and boosts the agent's initiative, ultimately offsetting the partial loss of control for the principal. In the case of Collab.Net's early activities, the congruence with the open source developers was obtained through the employment of visible open source developers (for example, the president and chief technical officer is Brian Behlendorf, one of the cofounders of the Apache project) and the involvement of O'Reilly, a technical book publisher with strong ties to the open source community.

Four Open Economic Questions about Open Source

There are many other issues posed by open source development that require further thought. This section highlights a number of these as suggestions for future work.

Which Technological Characteristics Are Conducive to a Smooth Open Source Development?

This chapter has identified a number of attributes that make a project a good or poor candidate for open source development. But it has stopped short of providing a comprehensive picture of determinants of a smooth open source development. Let us mention a few topics that are worth further investigation:

- *Role of applications and related programs*. Open source projects differ in the functionalities they offer and in the number of add-ons that are required to make them attractive. As the open source movement comes to maturity, it will confront some of the same problems as commercial software does; namely, the synchronization of upgrades and the efficient level of backward compatibility. A user who upgrades a program (which is very cheap in the open source model) will want either the new program to be backward compatible or applications to have themselves been upgraded to the new version.[37] We know from commercial software that both approaches to compatibility are costly; for example, Windows programmers devote a lot of time to backward compatibility issues, and encouraging application development requires fixing applications programming interfaces about three years before the commercial release of the operating system. A reasonable conjecture could be that open source programming would be appropriate when there are fewer applications or when IT professionals can easily adjust the code so as to ensure compatibility themselves.
- *Influence of competitive environment*. Based on very casual observation, it seems that open source projects sometimes gain momentum when facing a battle against a dominant firm, although our examples show open source projects can do well even in the absence of competition.[38] To understand why this might be the case (assuming this is an empirical fact, which remains to be established!), it would be useful to go back to the economics of cooperative joint ventures. These projects are known to work better when the members have similar objectives.[39] The existence of a dominant competitor in this respect tends to align the goals of the members, and the best way to fight an uphill battle against the dominant player is to remain united. To be certain, open source software development works differently from joint venture production, but it also relies on cooperation within a heterogeneous group; the analogy is well worth pursuing.
- *Project lifespan*. One of the arguments offered by open source advocates is that because their source code is publicly available, and at least some contributions will continue to be made, its software will have a longer duration. (Many software products by commercial vendors are abandoned or no longer upgraded after the developer is acquired or liquidated, or even when the company develops a new product to replace the old program.) But another argument is that the nature of incentives being offered open source developers—which, as discussed earlier, lead them to work on highly visible projects—might bring about a "too early" abandonment of projects that experience a relative loss in popularity. An example is the

XEmacs project, an open source project to create a graphical environment with multiple "windows" that originated at Stanford. Once this development effort encountered an initial decline in popularity, many of the open source developers appeared to move onto alternative projects.

Optimal Licensing

Our discussion of open source licensing has been unsatisfactory. Some licenses (e.g., BSD and its close cousin the Apache license) are relatively permissive, while others (e.g., GPL) force the user to distribute any changes or improvements (share them) if they distribute the software at all.

Little is known about the trade-off between encouraging add-ons that would not be properly supplied by the open source movement and preventing commercial vendors (including open source participants) from free riding on the movement or even "hijacking" it. An open source project may be hijacked by a participant who builds a valuable module and then offers proprietary APIs to which application developers start writing. The innovator has then built a platform that appropriates some of the benefits of the project. To be certain, open source participants might then be outraged, but it is unclear whether this would suffice to prevent the hijacking. The open source community would then be as powerless as the commercial owner of a platform upon which a "middleware" producer superimposes a new platform.[40]

The exact meaning of the "viral" provisions in the GPL license, say, or more generally the implications of open source licenses, have not yet been tested in court. Several issues may arise in such litigation: for instance, determining who has standing for representing the project if the community is fragmented, and how a remedy would be implemented (for example, the awarding of damages for breach of copyright agreement might require incorporating the beneficiaries).

Coexistence of Commercial and Open Source Software

On a related note, the existence of commercial entities living symbiotically off the efforts of open source programmers as well as participating in open source projects raises new questions.

The flexible open source licenses allow for the coexistence of open and closed source code. While it represents in our view (and in that of many open source participants) a reasonable compromise, it is not without hazards.

The coexistence of commercial activities may alter the programmers' incentives. Programmers working on an open source project might be tempted to stop interacting and contributing freely if they think they have an idea for a module that might yield a huge commercial payoff. Too many programmers might start focusing on the commercial side, making the open source process less exciting. Although they refer to a different environment, the concerns that arise about academics' involvement in start-up firms, consulting projects, and patenting could be relevant here as well. While it is too early to tell, some of these same issues may appear in the open source world.[41]

Can the Open Source Process Be Transposed to Other Industries?

An interesting final question is whether the open source model can be transposed to other industries. Could automobile components be developed in an open source mode, with GM and Toyota performing an assembler function similar to that of Red Hat for Linux? Many industries involve forms of cooperation between commercial entities in the form of for-profit or not-for-profit joint ventures. Others exhibit user-driven innovation or open science cultures. Thus a number of ingredients of open source software are not specific to the software industry. Yet no other industry has yet produced anything quite like open source development. An important research question is whether other industries ever will.

Although some aspects of open source software collaboration (such as electronic information exchange across the world) could easily be duplicated, other aspects would be harder to emulate. Consider, for example, the case of biotechnology. It might be impossible to break up large projects into small manageable and independent modules and there might not be sufficient sophisticated users who can customize the molecules to their own needs. The tasks that are involved in making the product available to the end user involve much more than consumer support and even friendlier user interfaces. Finally, the costs of designing, testing, and seeking regulatory approval for a new drug are enormous.

More generally, in many industries the development of individual components require large-team work and substantial capital costs, as opposed to (for some software programs) individual contributions and no capital investment (besides the computer the programmer already has). Another obstacle is that in mass-market industries, users are numerous and rather unsophisticated, and so deliver few services of peer recognition and ego gratification. This suggests that the open source model may not easily be transposed to other industries, but further investigation is warranted.

Our ability to answer confidently these and related questions is likely to increase as the open source movement itself grows and evolves. At the same time, it is heartening to us how much of open source activities can be understood within existing economic frameworks, despite the presence of claims to the contrary. The literatures on "career concerns" and on competitive strategies provide lenses through which the structure of open source projects, the role of contributors, and the movement's ongoing evolution can be viewed.

Notes

The assistance of the Harvard Business School's California Research Center, and Chris Darwall in particular, is gratefully acknowledged. We also thank a number of practitioners—especially Eric Allman, Mike Balma, Brian Behlendorf, Keith Bostic, Tim O'Reilly, and Ben Passarelli—for their willingness to generously spend time discussing the open source movement. George Baker, Jacques Crémer, Rob Merges, Bernie Reddy, Pierre Régibeau, Bernard Salanié, many open source participants, seminar participants at the American Economics Association annual meetings, European Economic Association Bolzano meetings, Harvard, and three anonymous referees provided helpful comments. Harvard Business School's Division of Research provided financial support. The Institut D'Economie Industrielle receives research grants from a number of corporate sponsors, including French Telecom and the Microsoft Corporation. This chapter is a modified version of Lerner and Tirole 2002, for which Blackwell Publishing holds the copyright and has granted permission for this use. All opinions and errors remain our own.

1. The media like to portray the open source community as wanting to help mankind, as it makes a good story. Many open source advocates put limited emphasis on this explanation.

2. It should be noted that these changes are far from universal. In particular, many information technology and manufacturing firms appear to be moving to less of an emphasis on basic science in their research facilities (for a discussion, see Rosenbloom and Spencer 1996).

3. This history is of necessity highly abbreviated and we do not offer a complete explanation of the origins of open source software. For more detailed treatments, see Browne 1999; DiBona, Ockman, and Stone 1999; Gomulkiewicz 1999; Levy 1994; Raymond 2001; and Wayner 2000.

4. Programmers write source code using languages such as Basic, C, and Java. By way of contrast, most commercial software vendors provide users with only object, or binary, code. This is the sequence of 0s and 1s that directly communicates with the computer, but which is difficult for programmers to interpret or modify. When the source code is made available to other firms by commercial developers, it is typically licensed under very restrictive conditions.

5. It should be noted, however, that some projects, such as the Berkeley Software Distribution (BSD) effort, did take alternative approaches during the 1980s. The BSD license also allows anyone to freely copy and modify the source code (as long as credit was given to the University of California at Berkeley for the software developed there, a requirement no longer in place). It is much less constraining than the GPL: anyone can modify the program and redistribute it for a fee without making the source code freely available. In this way, it is a continuation of the university-based tradition of the 1960s and 1970s.

6. Two main open source projects (GNOME and KDE) are meant to remedy Linux's limitations on desktop computers (by developing mouse and windows interfaces).

7. For example, Torvalds (interview by Ghosh 1998b) argues that the Linux model works best with developer-type software. Ghosh (1998) views the open source process as a large repeated game process of give-and-take among developer-users (the "cooking pot" model).

8. To be certain, backward compatibility efforts can sometimes be exerted by status-seeking open source programmers. For example, Linux has been made to run on Atari machines—a pure bravado effort, since no one uses Ataris anymore.

9. We focus primarily on programmers' contributions to code. A related field of study concerns field support, which is usually also provided free of charge in the open source community. Lakhani and von Hippel 2003 provide empirical evidence for field support in the Apache project. They show that providers of help often gain learning for themselves, and that the cost of delivering help is therefore usually low.

10. Linus Torvalds and others have been awarded shares in Linux-based companies that went public. Most certainly, these rewards were unexpected and did not affect the motivation of open source programmers. If this practice becomes "institutionalized," such rewards will in the future be expected and therefore impact the motivation of open source leaders. More generally, leaders of open source movements may initially not have been motivated by ego gratification and career concerns. Like Behlendorf, Wall, and Allman, the "bug fixing" motivation may have originally been paramount. The private benefits of leadership may have grown in importance as the sector matured.

11. Success at a commercial software firm is likely to be a function of many attributes. Some of these (for example, programming talent) can be signaled through participation in open source projects. Other important attributes, however, are not readily signaled through these projects. For instance, commercial projects employing a top-down architecture require that programmers work effectively in teams, while many open source projects are initiated by relatively modest pieces of code, small enough to be written by a single individual.

12. Dasgupta and David (1994) suggest an alternative explanation for these patterns: the need to impress less-informed patrons who are likely to be impressed by

the academic's undertaking research in a "hot" area. These patterns probably are driven by academic career concerns. New fields tend to be relatively more attractive to younger researchers, since older researchers have already invested in established fields and therefore have lower marginal costs of continuing in these fields. At the same time, younger researchers need to impress senior colleagues who will evaluate them for promotion. Thus, they need the presence of some of their seniors in the new fields.

13. Later in this chapter we will discuss *companies'* incentives to release code.

14. Open source projects might be seen as imposing less of a competitive threat to the firm. As a result, the firm could be less inclined to enforce its property rights on innovations turned open source. Alternatively, the firm may be unaware that the open source project is progressing.

15. To be certain, commercial firms (e.g., Netscape, Sun, O'Reilly, Transmeta) supporting open source projects are also able to compensate programmers, because they indirectly benefit financially from these projects. Similarly, the government and non-profit corporations have done some subsidizing of open source projects. Still, there should be an edge for commercial companies.

16. While we are here interested in private incentives to participate, note that this complementarity between apprenticeship and projects is socially beneficial. The social benefits might not increase linearly with open source market share, though, since the competing open source projects could end up competing for attention in the same common pool of students.

17. To be certain, commercial companies leave APIs (application programming interfaces) for other people to provide add-ons, but this is still quite different from opening the source code.

18. On the relationship between empowerment and career concerns, see Ortega 2000. In Cassiman's (1998) analysis of research corporations (for-profit centers bringing together firms with similar research goals), free riding by parent companies boosts the researchers' autonomy and helps attract better talents. Cassiman argues that it is difficult to sustain a reputation for respecting the autonomy of researchers within firms. Cassiman's analysis looks at real control, while our argument here results from the absence of formal control over the OS programmer's activity.

19. A standard argument in favor of open source processes is their massive parallel debugging. Typically, commercial software firms can ask users only to point at problems: beta testers do not fix the bugs, they just report them. It is also interesting to note that many commercial companies do not discourage their employees from working on open source projects. In many cases where companies encourage such involvement, programmers use open source tools to fix problems. Johnson (1999) builds a model of open source production by a community of user-developers. There

is one software program or module to be developed, which is a public good for the potential developers. Each of the potential developers has a private cost of working on the project and a private value of using it; both of which are private information. Johnson shows that the probability that the innovation is made need not increase with the number of developers, as free-riding is stronger when the number of potential developers increases.

20. An argument often heard in the open source community is that people participate in open source projects because programming is fun and because they want to be "part of a team." While this argument may contain a grain of truth, it is somewhat puzzling as it stands, for it is not clear why programmers who are part of a commercial team could not enjoy the same intellectual challenges and the same team interaction as those engaged in open source development. (To be sure, it may be challenging for programmers to readily switch employers if their peers in the commercial entity are not congenial.) The argument may reflect the ability of programmers to use participation in open source projects to overcome the barriers that make signaling in other ways problematic.

21. Valloppillil (1998) further argues that reaching commercial grade quality often involves unglamorous work on power management, management infrastructure, wizards, and so forth, that makes it unlikely to attract open source developers. Valloppillil's argument seems a fair description of past developments in open source software. Some open source proponents do not confer much predictive power on his argument, though; they predict, for example, that open source user interfaces such as GNOME and KDE will achieve commercial grade quality.

22. Open source projects have trouble attracting people initially unless they leave fun challenges "up for grabs." On the other hand, the more programmers an open source project attracts, the more quickly the fun activities are completed. The reason why the projects need not burn out once they grow in ranks is that the "fixed cost" that individual programmers incur when they first contribute to the project is sunk, and so the marginal cost of continuing to contribute is smaller than the initial cost of contributing.

23. E.g., Valloppillil's (1998) discussion of the Mozilla release.

24. It should be cautioned that these observations are based on a small sample of successful projects. Observing which projects succeed or fail and the reasons for these divergent outcomes in an informal setting such as this one is quite challenging.

25. The terminology and the conceptual framework are here borrowed from Aghion-Tirole 1997.

26. For instance, concerns about the "poaching" of key employees was one of the reasons cited for Steve Jobs's recent decision to cease giving credit to key programmers in Apple products (Claymon 1999).

27. For the economic analysis of employee visibility, see Gibbons 1997 and Gibbons and Waldman's (1999) review essays. Ronde 1999 models the firms' incentives to "hide" their workers from the competition in order to preserve their trade secrets.

28. Commercial vendors try to address this problem in various ways. For example, Microsoft developers now have the right to present their work to their users. Promotions to "distinguished engineer" or to a higher rank more generally, as well as the granting of stock options as a recognition of contributions, also make the individual performance more visible to the outside world.

29. Cusamano and Selby (1995), though, document a number of management institutions at Microsoft that attempt to limit these pressures.

30. Another motivation for commercial companies to interface with the open source world might be public relations. Furthermore, firms may temporarily encourage programmers to participate in open source projects to learn about the strengths and weaknesses of this development approach.

31. Red Hat provides support for Linux-based products, while VA Linux provided hardware products optimized for the Linux environment. In December 1999, their market capitalizations were $17 and $10 billion respectively, though they have subsequently declined significantly.

32. Of course, these programmers also increase the company's ability to learn from scientific and technical discoveries elsewhere and help the company with the development of the proprietary segment.

33. Companies could even (though probably less likely) encourage ex nihilo development of new pieces of open source software.

34. See, for example, the discussion of SGI's open source strategy in Taschek (1999).

35. It should also be noted that many small developers are uncomfortable doing business with leading software firms, feeling them to be exploitative, and that these barriers may be overcome by the adoption of open source practices by the large firms. A rationalization of this story is that, along the lines of Farrell and Katz (2000), the commercial platform owner has an incentive to introduce substitutes in a complementary segment, in order to force prices down in that segment and to raise the demand for licenses to the software platform. When, however, the platform is available through, for instance, a BSD-style license, the platform owner has no such incentives, as he or she cannot raise the platform's price. Vertical relationships between small and large firms in the software industry are not fully understood, and would reward further study.

36. An interesting question is why corporations do not replicate the modular structure of open source software in commercial products more generally. One possibility may be that modular code, whatever its virtues for a team of programmers

working independently, is not necessarily better for a team of programmers and managers working together.

37. The former solution may be particularly desirable if the user has customized last generation's applications.

38. Wayner (2000) argues that the open source movement is not about battling Microsoft or other leviathans and notes that in the early days of computing (say, until the late seventies) code sharing was the only way to go as "the computers were new, complicated, and temperamental. Cooperation was the only way that anyone could accomplish anything." This argument is consistent with the hypothesis stated later, according to which the key factor behind cooperation is the alignment of objectives, and this alignment may come from the need to get a new technology off the ground, from the presence of a dominant firm, or from other causes.

39. See, for example, Hansmann 1996.

40. The increasing number of software patents being granted by the U.S. Patent and Trademark Office provide another avenue through which such a hijacking might occur. In a number of cases, industry observers have alleged that patent examiners—not being very familiar with the unpatented "prior art" of earlier software code—have granted unreasonably broad patents, in some cases giving the applicant rights to software that was originally developed through open source processes.

41. A related phenomenon that would reward academic scrutiny is "shareware." Many of packages employed by researchers (including several used by economists, such as MATLAB, SAS, and SPSS) have grown by accepting modules contributed by users. The commercial vendors coexist with the academic user community in a positive symbiotic relationship. These patterns provide a useful parallel to open source.