The Future of Open Source

Ilkka Tuomi

Introduction

Open source has seen phenomenal interest and growth in recent years. In many ways, it has been a great success story. Clearly, it is no longer just hype, or a temporary fad. Yet it is interesting to think about the conditions that would enable the open-source movement to remain viable, and thrive. This chapter explores the driving forces behind this model, and the constraints on it, discussing both the factors likely to promote the continuous growth of the open-source movement and those that could lead to its downfall.

The sustainability of the open-source model depends on several factors. Some of these are internal to the model itself, including the economic viability of the model, the availability of competent contributors, and the extensibility and flexibility of the model. Other factors are external, including the potential reactions of proprietary-software developers and policymakers, as well as technological developments leading to evolutionary paths that are fundamentally incompatible with the model. Below I will discuss these factors, in an attempt to locate potential discontinuities that require new approaches from the open-source model if it is to maintain its vitality.

The history of open source

In the age of the Internet, new empires are rapidly built and lost. Successes quickly sow the seeds of their own destruction. The Internet, however, has proven to be extraordinarily flexible and capable of overcoming many of its inherent technical limitations, constraints and bottlenecks. This ability to innovate around emerging obstacles has been based on the distributed social model that underlies the evolution of the Internet and its key technologies. This distributed innovation model, in turn, is closely related to the phenomenon that we now know as the open-source development model. One might therefore expect that open-source software projects – and the open-source approach itself – would show similar viability and robustness when the time comes for them to reap the successes that they have sown.

The open-source software development model has been used since the first multi-user computers became available in the early 1960s. Robert Fano, one of the key architects of the first time-sharing system at MIT, described the phenomenon in the following words:

> "Some of the most interesting, yet imponderable, results of current experimentation with time-sharing systems concern their interaction with the community of users. There is little doubt that this interaction is strong, but its character and the underlying reasons are still poorly understood.
>
> The most striking evidence is the growing extent to which system users build upon each other's work. Specifically, as mentioned before, more than half of the current system commands in the Compatible Time-Sharing System at MIT were developed by system users rather than by the system programmers responsible for the development and maintenance of the system. Furthermore, as also mentioned before, the mechanism for linking to programs owned by other people is very widely used. This is surprising since the tradition in the computer field is that programs developed by one person are seldom used by anybody else… The opposite phenomenon seems to be occurring with time-sharing systems. It is so easy to exchange programs that many people do indeed invest the additional effort required to make their work usable by others." (Fano, 1967)

Fano further argued that a time-sharing system can quickly become a major community resource, and that its evolution and growth depend on the inherent capabilities of the system as well as on the interests and goals of the members of the community.

A system without a display, for example, could discourage the development of graphical applications, or if it were difficult for several people to interact with the same application this could discourage some educational uses. Moreover, Fano noted that after a system starts to develop in a particular direction, work in this direction is preferred and it accelerates the development in this direction. As a result, "the inherent characteristics of a time-sharing system may well have long-lasting effects on the character, composition, and intellectual life of a community" (cf. Tuomi, 2002: 86).

The modern concept of proprietary software emerged in the 1970s, when the computer-equipment industry began to unbundle software from hardware, and independent software firms started to produce software for industry-standard computer platforms. Over the decade, this development led to the realization that software was associated with important intellectual capital which could provide its owners with revenue streams. In 1983, AT&T was freed from the constraints of its earlier antitrust agreement, which had restricted its ability to commercialize software, and it started to enforce its copyrights in the popular Unix operating system. The growing restrictions on access to source code also started to make it difficult to integrate peripheral equipment, such as printers, into the developed systems. This frustrated many software developers, and led Richard Stallman to launch the GNU project in 1983 and the Free Software Foundation in 1985. Stallman's pioneering idea was to use copyrights in a way that guaranteed that the source code would remain available for further development and that it could not be captured by commercial interests. For that purpose, Stallman produced a standard license, the GNU General Public License, or GPL, and set up to develop an alternative operating system that would eventually be able to replace proprietary operating systems.

Although the GNU Alix/Hurd operating-system kernel never really materialized, the GNU project became a critical foundation for the open-source movement. The tools developed in the GNU project, including the GNU C-language compiler GCC, the C-language runtime libraries, and the extendable Emacs program editor, paved the way for the launching of other open-source projects. The most important of these became the Linux project, partly because it was the last critical piece missing from the full GNU operating-system environment. Eventually, the core Linux operating system became combined with a large set of open-source tools and applications, many of which relied on the GNU program libraries and used the GPL.

The first version of the Linux operating system was released on the Internet in mid-September 1991. The amount of code in the first Linux release was quite modest. The smallest file consisted of a single line and the longest was 678 lines, or 612 lines without comments. The average size of the files in the first Linux package was 37 lines without comments. In total, the system consisted of 88 files, with 231 kilobytes of code and comments. The program was written in the C programming language, which the creator of Linux, Linus Torvalds, had started to study in 1990 (Tuomi, 2004).

During the 1990s, the Linux operating system kernel grew at a rapid pace. The overall growth of the system can be seen in Figure 1. The accumulated number of key contributors recorded in the Credits file of the Linux system increased from 80 in March 1994, when they were first recorded, to 418 in July 2002, and 450 by the end of 2003. The developers were widely distributed geographically from the beginning of the project. In July 2002, there were 28 countries with ten or fewer developers and seven countries with more than ten developers. At the end of 2003, the Credits file recorded contributors from 35 countries (Tuomi, 2004).

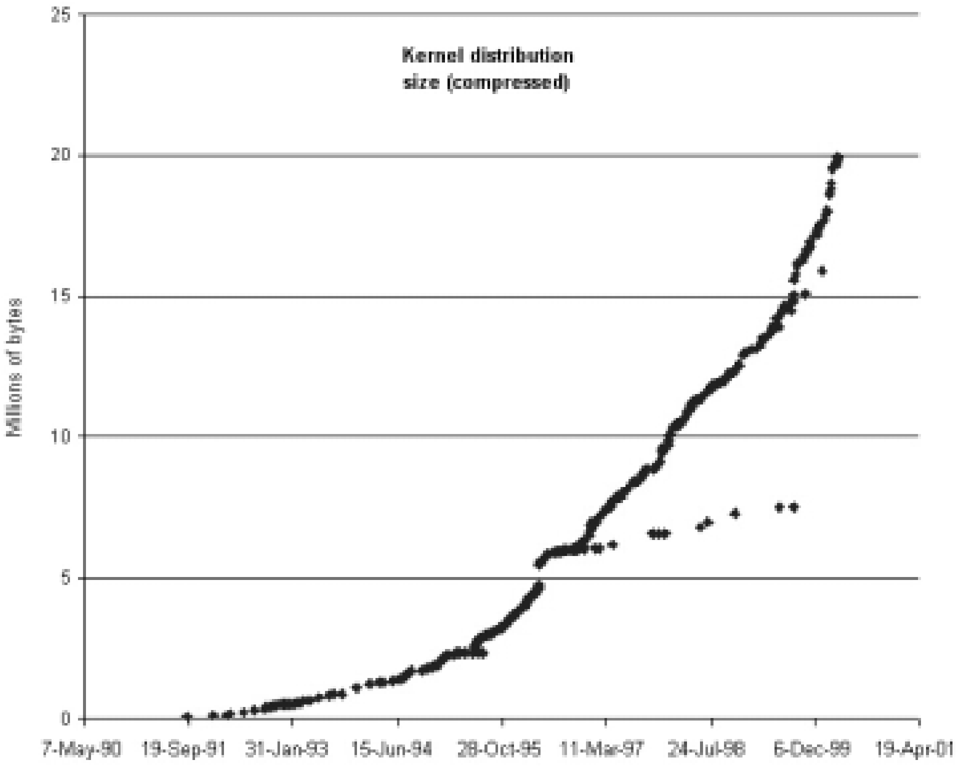

Figure 1 – Growth of Linux kernel, 1991-2000 (source: Tuomi, 2001).

Linux has become a particularly visible example of open-source software, as it has often been perceived as a challenger to Microsoft's dominance in personal-computer operating systems. Other important open-source projects, such as Apache, Perl, MySQL, PHP, Sendmail and BitTorrent, have also considerably shaped the modern computing landscape. In fact, the global Internet now operates to a large extent on open-source software. Commercial concerns, such as IBM, Sun Microsystems, Oracle, SAP, Motorola and Intel, have become important players in the open-source field. Policy-makers from South America to Europe, China, Republic of Korea, and Japan have become involved in open-source initiatives.

This widespread interest in open source, however, dates back only a few years. The breakthrough of open source to public consciousness occurred only after the turn of the decade. Although the first explorations of the open-source phenomenon appeared already in 1998, the first empirical articles started to become available in 2000.[1] The first policy-oriented studies began to emerge in 2001, when the public authorities in many European countries also became interested in open source.[2]

Driving forces behind the open-source phenomenon

There have been at least five major reasons for the growth of interest in open source. The first has been the cost. Open-source systems typically do not involve license fees, and users can download them from the Internet without paying for them.[3] Second, the open-source model has been claimed to produce better software than the proprietary closed-source model. The argument has been that the open-source development model allows multiple participants to peer-review effectively code developed by others. This has been claimed to lead to the fast development of high-quality systems. Third, open-source licenses and the availability of source code make it possible for users to modify the system to the specific needs of the user. Thus, if someone has particular, idiosyncratic requirements that will not be addressed by the producers of commercial software, the open-source model allows this end-user to extend the system so that it meets all his or her major needs. Fourth, the open-source development model has been claimed to lead to the faster incorporation of innovative ideas and new useful functionality than proprietary systems. This is because the distributed development model allows all the developers to contribute to the development of the system and, for example, to feed useful extensions to the system back to the developer community. Fifth, the availability of the source code enables users to check the functionality. This is expected to reduce the likelihood that the code may contain security vulnerabilities, such as back doors or malicious code.

In fact, these rationales for open source have rarely been carefully justified or studied. Proprietary-software developers have thus been able to make the counter-argument that, when the total lifetime costs for installing, operating and maintaining software are taken into account, the low cost of open source becomes questionable. In this argument, license costs are in any case a minor part of total costs.

The claims that open-source code is of better quality than proprietary code have also been mainly anecdotal, partly because very little has been known about the quality of individual open-source systems, or the open-source approach in general. Although its proponents have argued that the open-source development model leads to high-quality software, a quick glance at historical open-source releases typically reveals major quality problems even in the most successful of these systems.[4]

The argument that open source has more value for users than a closed system because open-source code can be modified does seem to have some historical justification. Successful open-source projects have grown because the end-users have been able to solve problems that they know well, and which are important to them. This, however, has typically meant that in successful open-source projects the "end-users" are themselves competent software developers. Successful open-source projects have an underlying social structure in which technology-producing communities substantially overlap with technology-using communities.

This, indeed, is the main difference between the proprietary commercial model and the open-source model. In the commercial model, users and developers typically form independent communities that are only indirectly connected through economic transactions. In the open-source model, by contrast, the development dynamic crucially depends on users who are also the developers of the technical system. Although there may be hundreds and even thousands of peripheral members in this community, the core community typically consists of a relatively small group of people. If these core developers stop the development, and no other developers take up their tasks, the system quickly dies away.

In this regard, many popular accounts of the amazing number of people involved in open-source projects have clearly exaggerated the size of open-source communities. It has often been argued that the success of the open-source model depends on thousands and even hundreds of thousands of community members. A more careful study of the nature of open-source communities, however, shows that, sociologically speaking, there is no such thing as "the open-source community", that almost all contributions to open-source projects come from a very small group of developers, and that hardly any open-source projects ever succeed in attracting the interest of more than a couple of developers or users.[5] In fact, a back-of-envelope calculation of the total resources used to develop Linux – the flagship of the open-source movement – indicates that, on average, it has been developed by the equivalent of perhaps a couple of dozen developers per year.[6] This is by no means a trivial software development project, but it could hardly be called an extraordinarily large development effort.

The argument that the open-source model would lead to more innovative systems, and the faster incorporation of new technological ideas, is an interesting one. It also leads, however, to the difficult and important question of what, exactly, we mean by innovation. At first sight, there is nothing particularly innovative in projects such as Linux, which basically re-implements commercially available operating-system functionality. Software engineering, in general, is engineering: the implementation of a specific, given functionality using commonly accepted tools and methods. Innovative operating-system architectures exist, including the Unix system architecture, on which Linux is based, was innovative in the early 1970s, when it was first developed. Instead of characterizing the Linux project as an innovation project, one might therefore be justified in arguing that it is more accurate to view it as an engineering and implementation project.

If, however, by innovation we mean all kinds of technology development, a detailed study of the evolution of open-source projects shows that they structure innovation processes in very specific ways. In the case of Linux, for example, the social control of technology development has become tightly aligned with the modularity of the technical system architecture. Some parts of the system were frozen and excluded from modifications in the very early stages of the development. These stable, core elements of the system have, in turn, allowed the rapid expansion of the system in the more peripheral areas. Indeed, almost all of the growth seen in Figure 1 results from code that has been added to link the core operating system with new hardware and peripherals. This can se seen in Figure 2, which shows the incremental code changes in one of the core modules of the Linux operating system – the kernel module – and one directory path consisting of code that includes some of the main extensible parts of the system.

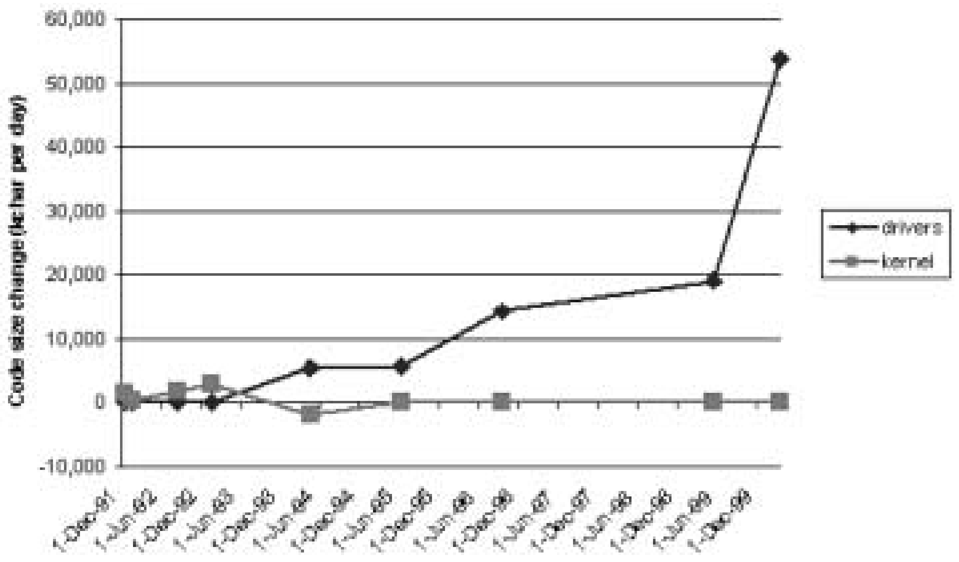

Figure 2. Growth of source code in two main Linux code directories (data from Tuomi, 2002).

This model of development has the social advantage that the developer community can absorb enthusiastic new developers without any great difficulty. The new developers can learn their skills and work practices by developing code that extends the system's functionality but does not interfere with its core functionality. Gradually, the novices can then earn a reputation as reliable developers, and become masters and gurus in the project community.

This process of social integration and skills development is closely related to the architecture of the technical system that is being developed. Not all systems can be built in a modular and incremental fashion that simultaneously supports skills development, the socialization of community members and the development of a functional technological artifact.

The roots of Linux's success can be found in the architectural features of Unix. The main architectural insight contributed by the Unix system was its modular structure, which allowed new functionality to be added to the system simply by adding on new program modules which used the existing resources to get the job done. Historically, the GNU project relied on the fact that a full replacement for the commercial Unix operating system could be built piecemeal, by using the existing Unix platform as the development platform. The GNU project, therefore, could start to replace the proprietary components of the Unix system one by one, and test the functionality of the newly developed open-source modules in the context of the existing system. In this sense, the GNU project tried to keep the functionality of the GNU system compatible with the existing Unix, while incrementally reworking the commercial Unix in the direction of an open-source version. Indeed, the GNU project gradually built almost a complete system, which finally became a full operating system when the Linux developers plugged in the missing piece, the operating-system kernel.

The skills development model in GNU/Linux closely resembles the social learning processes that underlie the dynamic in communities of practice. Such social learning processes have already been studied for a long time outside the software-development domain. Lave and Wenger (1991) proposed a model in which communities of practice form the social entities that maintain particular social practices and stocks of knowledge, and Brown and Duguid (1991) extended this model to organizational innovation models. Historically, these models were based on ideas developed as part of the cultural-historical activity theory, which emphasized the development of skills and cognition as a process in the "zone of proximal development" where the performer was able to show competent behavior when supported by others, but where autonomous performance was still lacking.[7] By borrowing skills and knowledge from others, the learner was able to do things that still remained outside his or her current capabilities and, through such socially supported action, learn them. In this model, parents, for example, built mental and practical "scaffolds" that allowed their children to climb to new levels of performance, where these new, advanced forms of cultural and cognitive behavior could be experienced and learned, and eventually performed without support. In technology-development communities, a similar process of skills development leads to social structures that consist of "novices", "old-timers" and "gurus", and which center around community-specific knowledge and artifacts (Tuomi, 1999).

The problem inherent in this model is that it is fundamentally a conservative one. A commercial Unix system can be transformed into an open-source version piece by piece only if most pieces are kept stable at every point in time. As depicted by Lave and Wenger, the "community of practice" model was focused on the transfer of existing social practices and traditions. The power structure was concentrated in the "center", with the members of the core community defining what counted as knowledge and development in the community in question. The novices could enter the community gradually, by first gaining access to the community, then internalizing its values and world-views, and eventually becoming full, competent members.

This community-centric developmental model, therefore, could be expected to be particularly suitable for incremental innovations and extensions that enforce the basic values of the community. In this model, it would be quite difficult to introduce radical innovations that shake the current power structure or contradict the core values of the community. For example, if the basic hardware architecture of personal computers were to change radically, such a change might require extensive changes to the structure of the Linux developer community, possibly leading to the end of Linux.

The fifth reason for the increasing visibility of open source – better security through the availability of the source code for inspection – has gained some importance in recent years. In general, computer viruses, daily announcements of security problems in commercial software and frequent hijackings of network-connected computers in order to relay spam have pushed security to the forefront of users' concerns.[8] Computer users have realized that their broadband-connected computers are accessible across the globe, and policymakers have become worried that unauthorized access could lead to problems with national security and computer crime. Well-publicized US initiatives on monitoring and tracking electronic communications and the lack of transparency of government efforts to fight terrorism have also increased the perceived possibility that commercial software vendors may be required to incorporate unpublicized back doors to their systems to permit and facilitate the monitoring of computer-based activities around the world.

If the commercial software producers cannot be trusted, or held liable for damages, the only alternative is to establish an independent control process that guarantees that problems do not exist. At first, it might perhaps look as if the availability of the source code would solve this problem. If the code can be inspected, then in theory the inspectors can easily see a code that implements back doors or other unacceptable functions. In practice, this is more difficult, as can be seen from the constant flow of security-related updates for both closed-source and open-source software. The inspection of software code is relatively easy for the people who have been involved in the development of the system, but it is difficult for people who are given source code that represents perhaps years of accumulated work.

More fundamentally, however, the source code does not itself reveal all security problems. When the system is compiled into a working binary code, the compiler decides how the transformation of the source program code is done. If the compiler, for example, instructs the programmed system to use code that includes a back door, all systems compiled with this compiler will include the back door. Similarly, if the microprocessor microcode, which the program eventually mobilizes to do its tasks, includes undocumented instructions, no amount of inspection of the program source code will reveal the related security problems. Such microcode instructions could be used to bypass security mechanisms implemented in the open-source program.[9]

Although it is clear that the availability of source code does not completely solve the problem of security or lack of trust, it does, however, considerably limit the types of problems. Only microprocessor manufacturers can implement microcode on their chips, and only compiler-developers can change the basic functionality of compilers. The question still remains, however, whether the open-source approach leads to better security than closed source. The argument for closed source has been that keeping the source code unavailable makes it more difficult for hackers and computer criminals around the world to develop malicious code. This "security through obscurity" argument, however, has been discredited by most security experts.

The proper argument would be that the developmental path of open-source leads to systems that have different security characteristics than closed source systems. This could happen, for example, because security problems were detected faster in the open-source model, as more people were able to study the code, or because the availability of source code enabled the developers to build mental models of the system architecture, system functionality and accepted development styles that facilitated the detection of anomalies. It could also happen because existing security problems were widely publicized within the development community, thus enabling its members to learn faster how to develop code without security problems. The open-source model could also lead to better security simply because the transparent development model makes the individual developers personally and socially liable for the quality of code.

All these reasons, of course, depend on the poor quality of commercial-software development processes. In principle, one could invest in improvements of commercial software development processes to make them start to produce software as good as the open-source model. In theory, however, one might also argue that the open-source model is inherently better than any closed-source models in addressing security problems. Such a claim could be based on two separate points. The first is that the open-source model allows users to organize independent quality-control mechanisms. The second is that distributed development is inherently better, and that it requires open access to the source code.

Independent quality controls and the possibility of inspecting code are important if the quality-control mechanisms of commercial software developers cannot be trusted. Normally, in such a situation, commercial partners try to manage the risks by agreeing on liability and remedies in case damage occurs. In practice, however, this is an option only when the risks are relatively small or controllable, for example by insuring against them. In reality, many producers of prepackaged software and operating systems would probably go bankrupt if they had to cover all the damage and loss generated by quality problems with their products. For this reason, software vendors typically only license the right to use their products under conditions where they are not liable for any damage caused by such use. In this situation, the possibility of reviewing vulnerabilities and defects independently, and reacting to them, does have some value for users.

The second argument for the inherent security of open source would be that distributed and self-organized development processes always win in the end, even over the best-organized development processes. This argument would be akin to the Hayekian view that market-based economies are always better than command economies because they allocate resources more effectively and process information better than any hierarchical decision-maker. One could argue, for example, that detecting and solving security problems requires local, context-specific knowledge that always remains inaccessible to central decision-makers, who can therefore never make optimal choices.

This Hayekian story, of course, would not be the full story. Taken to its extreme, it would leave open the question of why organizations exist. Indeed, this question is also central to the future of open source.

The economic organization of open source

From the point of view of economic theory, the open-source development model is a challenge. Much of the literature on economics starts out from the assumption that economic players need to appropriate the returns of their investments. In this theoretical world, economic players produce new technology only if they can make a profit. Furthermore, if the developers are unable to perceive and appropriate all the benefits of their development, investment may remain below the social optimum. As complex production requires the division of labor and capital, entrepreneurs set up business companies. Business organizations, therefore, emerge as the principal actors on the stage of modern economy, and, in this theoretical framework, need to become owners of the products they produce.

The concept of ownership is a central one in modern society. It translates the social phenomenon of power into the domain of ethics, rights, and legal institutions. In practice, the concept of ownership makes it possible for people to exchange valuable things and services in an orderly and predictable way, without relying on pure and random violence. We can always take the goods we need if we are powerful enough, but if the power is physical instead of social, the behavior can appropriately be called asocial. Ownership stops us acting purely individually, as we cannot fulfill our own needs without asking who the others are who control goods and resources, and what they want. In this sense, the concept of ownership is the foundation of social worlds and ethical behavior. The acceptance of ownership structures means that we have tamed our nature as beasts, and accepted the structures of society.

In open-source communities, however, the concept of ownership becomes redefined. Instead of controlling a given good, open-source communities control the developmental dynamic of an evolving good. The “openness” of open source, therefore, is more about open future than about access to currently existing source-code text.

Simple extensions of conventional economic concepts, such as ownership and intellectual property, are therefore bound to create confusion when applied in the context of open-source development.

In economic discussions on open source it has frequently been argued that open source may be understood as a public good. In economic theory, public goods are goods that are non-exclusive and in joint supply. A feature of non-exclusive goods is that if they are available, they are available to all. Public goods that are in joint supply, in turn, are goods that do not lose value when someone benefits from them. Open source fulfills both conditions: when it is made available, it can be downloaded by anyone, and when someone downloads the system, its value for other users does not decrease.[10]

Von Hippel and von Krogh (2003) have argued that the open-source model works well because the developers have sufficient private benefits to keep the development going. The traditional theoretical problem with the production of public goods is that when the good is available to everyone, those who have not contributed to its production can also benefit from it. This leads to free-riding and less-than-optimal investment in development. For this reason, economists often believe that in well-operating markets all production should be private production where the producer is fully entitled to maximize the profits from his or her investment. The von Hippel and von Krogh argument was that free-riding does not necessarily have to be destructive. The developers may gain access to valuable learning and system functionality that is tailored to their specific needs, whereas the free-riders can only benefit from the system itself. If the private benefits to developers are sufficient, the open-source model can produce public goods without the risk that all developers may end up free-riding on the work of others. The apparent miracle of open source can therefore be compatible with the established beliefs of economic theory. More specifically, the miracle is shown to be an illusion, which reveals its true nature when the private benefits of the benefit-maximizing individual developers are taken fully into account.

O'Mahony (2003) pointed out that although open source may be a privately produced public good, one still has to consider the conditions that make it difficult to steal this good. In particular, she highlighted the different tactics that open-source communities have used to keep the system they have produced a public good. These include the open-source licensing terms and branding that restrict the possibility for commercial players to extend the code to make it, effectively, proprietary.

O'Mahony also noted that open-source software could be described as a common-pool resource. In economic theory, common pools have been used to analyze tragedies of commons, where individual players maximize their benefits to the eventual loss of everyone. Traditional discussions on common pools and tragedies of commons assumed that economic players are norm-free maximizers of their immediate individual benefits, without the capacity to cooperate. Empirically, these assumptions are obviously wrong when applied to open-source communities. Since the foundation of the GNU project, the explicit goal of many open-source projects has been to collaborate in the production of common goods, and history shows that they have successfully done exactly that.

A feature of common-pool resources is that they are subtractable. In other words, when someone uses the pool, its value diminishes. This, in fact, is a common feature in real life if resources are not renewable or if their renewal occurs more slowly than their depletion. From this point of view, open source is an interesting resource. As the future value of the system depends on the amount of developers and the availability of complementary products, the open-source pool may in fact become more valuable the more people use it. In this sense, open source could be described as a fountain of goods. Traditional economic theories have difficulty modeling such phenomena, as they are built on the assumption that resources are scarce. In "fountains of goods" models, the limiting factor for growth is not the decreasing marginal benefit and increasing cost; instead, it is to be found in the dynamic of social change and the development of the skills needed to seize the emerging opportunities. In other words, such models require that we move beyond traditional economic theory.

Economists have typically tried to show that existing theoretical models can be compatible with the open-source model when purely economic transactions are complemented with concepts such as investment in reputation and network effects. Lerner and Tirole (2000) highlighted the potential importance of delayed payoffs, such as enhanced career opportunities and ego gratification generated by peer recognition. Johnson (2001) showed that open-source development may be modeled as a simple, game-theoretic model of the private production of a public good, where the developers optimize their benefits under the conditions of perfect knowledge about the preferences of others. Dalle and Jullien (2001), in turn, showed that network externalities could make software users switch from proprietary systems to open systems, modeling Linux users essentially as a magnetic material which jumps from one organized state to another depending on external forces and the interactions between its nearest neighbors.

In fact, many early models were quite independent of the empirics of open source, simply because relatively little was known about open source. The models could easily have been generalized to fit practically any economic activity where network effects, interdependent investments, or delayed benefits would have been relevant. Economic theory, however, has usefully highlighted the essential economic characteristics of open source. It can be understood as a privately produced good, in the sense that individual developers create contributions to the system that creates public benefits. The developers make decisions about joining development projects based on their perceived benefits and costs. The benefits may include enhancement of reputation, the value of developed functionality to the developer, peer recognition, and other short-term or long-term benefits. When the individuals are working for commercial firms and paid for their work, they may also develop the system because they are paid for it. Although it is not clear whether the developers maximize their benefits in any systematic sense, it is clear that they do take them into account. In addition to straightforward economic benefits, however, open-source developers consider a broad set of possible benefits, including the excitement of being part of a meaningful social project that can potentially change the world.

In other words, open-source developers are motivated and incentivized in many ways and for many different reasons. The particular strength of the open-source model is that it allows multiple motivational systems to co-exist and to be aligned so that the system development goes on. In open-source development, in particular, it is not only money that counts. In this economy, sellers, buyers, and producers may market things for example because they like being on the market, and because the game of social interaction is fun and socially meaningful.

Conventional economic thinking has had conceptual difficulty in dealing with open source because economic theory has historically centered on scarce resources. The basic historical problem for classical economists has been how to maximize the consumption of scarce goods. This was a relevant question in a world where the lack of consumption possibilities was a daily challenge and where people lived in poverty and hunger. Open-source development, in contrast, is a social process that creates goods where they did not exist before. The concept of scarcity cannot easily be applied when the economy becomes creative. The open-source development model, therefore, shakes one of the core building blocks of the modern economic worldview.

In fact, some of the excitement in open-source communities seems to result from the realization that open source stands the basic principles of the modern global economy on their heads. In open-source projects, consumers become producers. Instead of becoming alienated from the results of their work, open-source developers engage in individually and socially meaningful production, and retain the moral authorship of their work. In this sense, the social currents that express themselves in open-source projects can also be viewed as a positive and productive version of anti-globalization critiques, for example.

Perhaps the most obvious – if somewhat controversial – reason for the inability of conventional economic theories to describe the open-source phenomenon is that economic transactions and economic rationality operate only in limited social domains. A modern economy has a very particular way of organizing social interactions. In its present form, its history in industrialized countries goes back only a couple of centuries. Many areas of social life, therefore, remain outside the sphere of economic models and conventional economic theorizing.

Most fundamentally, perhaps, the concept of rational choice that underlies microeconomics does not work well in innovative worlds. In practice, people are able to revise their priorities and perceptions rapidly, and in the modern world, where new social and technical opportunities emerge frequently, such revisions are common. In effect, this means that people change the reasoning behind their actions and their preferences in ways that make traditional economic models unable to predict social behavior, except when social change is of minor importance. In new technology development this is rarely a good approximation. The conceptual structure of traditional economic models requires that the rules of the game remain essentially constant and that the world of preferences is closed. In technology development, however, innovation produces essentially novel social phenomena, opens up new worlds for social action, and constantly changes the rules of the game. This makes it necessary for individuals and other economic decision-makers to re-evaluate and reinvent their value systems continuously.

Economic theories normally cannot handle such situations, as, methodologically speaking, they rely on an implicit empiristic and positivistic epistemology. In other words, they assume that stable value systems exist, and that economic behavior consists of expressions of individual preferences within objective value systems. The lack of stable, objective value systems in society means that conventional economic theories do not, in general, converge towards any social reality. Economic explanations of technology-creation activities, such as open-source development, therefore remain abstractions lacking predictive power.

This theoretical challenge, of course, does not mean that concrete economic factors are irrelevant to the future of open source. In fact, both the sustainable evolution of community-specific value systems and the accumulation of traditional economic resources are critical for the success of open-source projects. These projects are interesting for economic theory, as they highlight the importance of community-based value systems and social structures at the foundation of an economy. The substructure of a modern economy is revealed in these technology-development projects, which organize themselves for the purpose of collective production and social exchange. The particular characteristics of modern, monetarized economic transactions, in turn, become visible when open-source projects need to interface their activities with the rest of the modern economy.

The basic challenge for sustainable social activities is that they need to generate and accumulate sufficient resources to maintain themselves. In conventional economics, this requirement has often been understood as the need to generate profit. Businesses are viable if they can pay for their activities and investments and pay their investors.

In the case of open-source development, early descriptions of the phenomenon often argued that the developers worked completely outside the economic sphere. This, of course, was never the case. Richard Stallman, for example, explicitly pointed out in the 1980s that he generates income for the GNU project by charging for the distributions of the system, that he makes the work on the project possible by commercial consultancy and teaching, and that the continuity of the project depends on donations of money and equipment. Open-source developers have used technology developed by commercial firms and computer networks funded by universities and governments, and they have often had to pay for their pizzas in real currency. In this sense, open-source development can also be understood as a parasitic activity that has been free-riding slack resources and activities within the sphere of economic transactions. In fact, the tradition known in French as *la perruque*, whereby workers skillfully transfer their employer's resources into gifts or meaningful and unintended productive activities, could easily be used to explain some characteristics of the open-source economy (Tuomi, 2002:28).

A more careful look at the parasitic nature of open source, however, reveals that it is not obvious who the parasite is and who is the host. All economic players – including commercial software vendors – free-ride extensively. For example, they rely on skills development processes that are paid for by others, often by the workers, competing firms, or the public schooling system. Public and private investment in the Indian university system, for example, allow many commercial software firms to lower their expenditure. Non-commercial software activities, such as hobbyist development, computer gaming, demo development, and various other forms of computer hacking have provided critical sources of skills for commercial vendors at least since the 1970s. Innovative commercial activities paid for by competitors have also been important. The history of Microsoft is an illustrative example here.

It may therefore be claimed that open-source developer communities have been possible because they have successfully appropriated resources without paying for them. It may also be said, however, that commercial software vendors have been possible for the same reason. Instead of one or the other being a parasite, these two modes of software production have been living in symbiosis for a long time. The question for the viability of the open-source model, then, is whether the growth of the open-source movement disturbs this symbiosis in major ways, or whether open-source communities may, for example, jump to a new symbiotic relationship where commercial software vendors are no longer needed for open-source development.

Recent developments have shown that both commercial vendors and open-source communities are transforming themselves in order to address this challenge. New hybrid models are emerging in which open-source communities and commercial profit-making firms redefine their relations and explore mutually viable models. MySQL, for example, has successfully developed a Janus-like model where it shows one face to the open-source community at large and has a different, commercial interface available for profit-making players. Several Linux distributors have developed business models that combine value-added services with open-source community models. IBM and other profit-making firms have internalized some open-source activities with the aim of developing a competitive advantage over closed-source competitors.

The politics of open source

Experiments with hybrid models that combine community-based open-source development with profit-making organizations will show what types of symbiosis are possible. For the open-source community, however, there is also another alternative. If open source is, indeed, a public good or a public fountain of goods, it could also legitimately be supported through public policy.

The challenge of public policy, in general, is that if it is effective it changes the world. Successful policy, therefore, by definition, cannot be neutral. In particular, if public policy supported open-source activities, it would implicitly reorganize the field in which software producers operate. When the assumption is – as it commonly is – that policy should not interfere with market forces, all policy changes that have a market impact appear problematic. For example, if public resources were used to fund open-source software production, the producers of closed systems could claim that such behavior threatened free competition. More importantly, if public authorities required open-source licenses for publicly procured systems, they could easily be viewed as interfering with competition.

The European Union has a particular challenge in this regard. National policies can often override or balance competition concerns when other policy objectives – such as wars against terror, national security, or industrial and economic development – are involved. In the European Union, however, the legal basis for action is to a large extent built on the assumption that competition in a free market is a priority. This apparently neutral approach is not as neutral as it may seem at first sight. Free competition is not something abstract or absolute: it can only exist within given institutional structures. Any change in these institutional structures, therefore, necessarily interferes with the current state of competition. In this sense, the idea of free competition is inherently conservative, and it often quite strongly influences existing and established interests.

Policy intervention with a clear market impact has, therefore, often focused on the particular cases where theoretically accepted reasons allow policymakers to claim that markets have failed to operate as they should. As a consequence, much of the competition policy in free-market societies has centered on antitrust issues and monopolies.

The open-source phenomenon opens up new challenges for policy. For example, it is possible to argue that a feature peculiar to the dynamic of the software industry is that, in this domain, positive returns, first-mover advantages, and scale-effects of production generate an industry structure that is socially and economically not optimal. New software is often adopted in environments where interoperability is important. After new systems have been integrated into existing environments, switching costs may become very high, and in practice users may therefore be stuck with the choices they have made in the past. Producers of new, improved software may therefore find it impossible to survive in this competitive landscape, where entry-points are tightly controlled by existing players. The end result might be, for example, that the structure of the software industry will evolve rapidly into one where those first in are the dominant players, and where small newcomers have great difficulty in growing. From a policy point of view, this could mean, for example, that fewer jobs and services are produced than would be the case if the rules of competition were balanced by policies that mitigated the excessive impact of network effects and first-movers' historical advantages.

One could also argue, for example, that wider use of open-source software would make it easier to find socially beneficial uses for new technology. When a society becomes increasingly computerized and many of its systems depend on software, the transparency of these systems can give entrepreneurs and innovators the chance to see where their contributions might create value. This, essentially, is how the existing open-source projects have become successes. When competent developers can access the source code and see what functionality could best extend the capabilities of current systems, they can maximize the impact of their work. One might expect that the benefactors of such improvements would often be happy to reward this development work, thereby also providing commercial opportunities for software developers. In policy terms, therefore, open source could lead to demand creation, economic growth, and jobs.

These simple examples highlight the point that policies relating to software access and openness may have serious social and economic consequences. Software is becoming one of the main economic factors in society. Software-related policy issues are therefore bound to become increasingly important. In the future, we may, for example, need a better understanding of the development dynamic in software industries, as well as new concepts of system innovation and interoperability that will allow policymakers accurately to define policy issues. Research on open-source projects will provide useful insights on these challenges.

The future

The history of open source indicates some factors that will also be important for its future. As was noted above, a critical characteristic of the open-source model has been the ability to integrate new community members and to support them effectively with social learning models. The access to competent community members, community discussions and historical records has played an important role here.

This collaborative learning model could potentially be extended beyond software development projects. For example, it could provide a foundation for community-centered social innovation projects, where simultaneous knowledge creation and skills development are important (Tuomi, 2003). This, in itself, could generate global demand for open-source tools that support such processes.

A unique characteristic of software is that the description of the system is also the system that is developed. In this domain, the technical artifact and its specification coalesce into one. When developers have access to the source code, their technical skills and knowledge about the functionality of the system can therefore be developed with great effectiveness. For example, when the developers talk about specific problems within the current system, they refer not to abstract descriptions of the system but to the system itself. Alternative solutions to problems can simply be compiled into binary code and run on computer hardware to resolve different opinions about the functioning of code.

This also makes open-source software development different from science. In the empiristic scientific tradition, the idea was that abstract theories could be tested by comparing their predictions with observable natural phenomena. Given an objective, observer- and theory-independent reality, theories were expected to converge eventually towards an accurate description of this reality. As philosophers of science have pointed out, this project was, in itself, unrealistic. The relevance of observations depends on the theories used, and theories depend on historically accumulated conceptual systems. In open-source projects, the dream of objectivistic knowledge, however, is approximately true. As long as the underlying technical architecture stays unchanged, it works as an objective, external world, against which different theoretical models and abstractions can be tested.

When the underlying technical architecture changes, the impact of this depends on the nature of the change. If it affects only minor features, many of the previous abstractions remain valid. If the changes are radical, the abstractions may need to change radically.

In the world of open source, radical changes could be generated, for example, by innovative new hardware architectures that require completely new approaches to design. For example, if Intel suddenly switched its microprocessor architectures to support parallel message-passing, quantum computing, or new adaptive information-processing approaches, the Linux operating system community might have great difficulty in adjusting its designs and its internal social structures.

The history of open source also highlights the importance of developer motivation and needs. Open-source development has often met the needs of the developer, and addressed the frustrations generated by constraints generated by others. If, for example, Xerox had given access to the source code of its printer drivers, and if AT&T had not restricted the sharing of Unix among universities, it is quite possible that the GNU project would never have been launched. When commercial software vendors increasingly make their source code available, therefore, some of the motivation for launching open-source projects disappear. Indeed, this “embrace of death” strategy is probably part of what underlies some current commercial open-source initiatives.

Open-source developers have also often pointed out that they develop software because it is fun. If software development were no longer fun, open-source projects would look less attractive. For example, if programming tools become so advanced that useful applications could easily be developed by anyone, many of the low-hanging fruits of software development could be picked by people who are not particularly interested in technical challenges.

Some technical and policy changes could also lead to dead ends in open-source development. One widely discussed issue has been the conflict between open-source licenses and legal restrictions to reverse-engineer or publish code and algorithms that implement security or digital rights management. Open-source licenses require code to be distributed in a form that allows its modification. If the system, for example, interfaces with commercial products that protect against unauthorized copying, the copy protection algorithms may need to be published whenever the system or its modifications are distributed. Or if the system has encryption or privacy characteristics that cannot be exported without government permission, the system may be incompatible with open-source principles. Similarly, in the future some governments might require all operating system to include back doors for crime enforcement. Such requirements could make open source impossible or illegal in its present form.

The ingenuity of the original GNU license was that it applied recursively a very generic constraint that guaranteed the growth of the system. The name GNU itself was a play on the principle of recursion, an acronym coming from “GNU is Not Unix.” The C language that was used to program Unix is often used in this recursive way, where program subroutines or functions iteratively use themselves to compute complex programs in a simple and compact way. The GPL is a similarly recursive program. It keeps the constraints of development constant for all unforeseeable situations, as long as the development goes on. In this sense, the GPL license was a nice, clean example of good programming. This time, however, it was social engineering: it programmed social behavior, instead of computers.

Many modified open-source licenses lose this basic characteristic of the GPL. At the same time, they allow a partially recursive process of open-source development to go on as one specific developmental branch. The growth constraints and viability of this open-source branch, however, are no longer defined from the start. Instead of having only the choice between growing and dying, the more restricted forms of open-source licenses can also lead to privatized developmental paths and the withering of the original open-source project. This is exactly the reason why additional social procedures, such as those discussed by O'Mahony, become necessary. If open-source pools can be converted into private wells or private fountains, branding and social sanctions such as excommunication from the programmer community may become necessary. Such social strategies, however, are rarely foolproof. If pool conversion is possible in principle, in practice it may crucially depend on incentives and on the possibility of offsetting the immediate damage generated during the conversion. If the price is high enough, open-source developers may happily sell their fountains, even though they might afterwards find themselves running out of water. A similar issue underlies much of European thinking of privacy regulations and consumer protection, which often starts from the assumption that individual bargaining power may sometimes need policy support, and that some types of economic transactions should not be allowed on free markets.

One interesting challenge for future open-source development is also the issue of liability. Commercial software vendors are able to make contracts that free them of all liability. As open-source developers typically form open and undefined communities, usually there are no institutionalized agents that could make contracts for open-source systems. Modern legal systems simply do not acknowledge the existence of such open, productive communities. Furthermore, the success of open source greatly depends on the fact that its users and developers do not have to sign contracts with all the people who have contributed to the code.

This opens up a loophole for competitive strategies that use the historical institutions of the law as a weapon against new forms of organizing and acting. Relying on the institutional blindness of justice, closed-source software vendors can make open-source systems unattractive to established institutions. This, of course, is exactly what has happened recently with Linux and its competitors, such as SCO. As another example, in mid-2004 a think-tank from Washington, D.C., published a report alleging that the Linux kernel code was probably borrowed or stolen from an earlier Minix system. The fact that there was no systematic documentation on the history of Linux code, and no simple way to trace the sources of all the various contributions, made it possible to argue that someone might eventually sue Linux users for damages.[11] As under the legal system in the US it is possible to claim very large punitive damages, this risk could obviously be a problem for Linux users operating there. Similar approaches, where existing legal systems are used as competitive weapons, could potentially slow down or kill open-source projects in the future. For the viability of the open-source model, it might, therefore, be necessary to develop liability rules that limit the possibility of misusing such competitive approaches.

The final challenge for the open-source model is its ultimate success. Many open-source developers have built their identities around a project that has been designed to resist the hegemony of the dominant software giant, Microsoft. If Linux one day succeeds in conquering servers and desktops around the world, this basis for resistance will evaporate.

In the Hegelian explanation of world dynamics, development is driven by contradictions. Success sows the seeds of its own destruction, and revolutions eat their children. Before this happens it, however, open source may become normal. Revolution may turn into evolution. Open-source communities may traverse the historical phases of social development in Internet time, finding again the traditional forms and problems of community, organization, economy; and eventually moving beyond them. Commercial developers may perhaps become open-source developers, as the software industry finds new forms of synthesis, reconciliation, and symbiosis. This, exactly, is why open source can survive: the future is open.

Notes

* The views expressed in this chapter do not represent the views of the Joint Research Centre, the Institute for Prospective Technological Studies, or the European Commission.

1 The early papers included several papers published in First Monday including Ghosh, 1998; Bezroukov, 1999; Kuwabara, 2000; Ghosh & Prakash, 2000; Edwards, 2000; Moon & Sproull, 2000; and other important contributions such as Kollock, 1999; Dempsey, Weiss, Jones, & Greenberg, 1999 Mockus, Fielding, & Herbsleb, 2000; Koch & Schneider, 2000; Feller & Fizgerald, 2000; Ljungberg, 2000; and Yamauchi, Yokozawa, Shinohara, & Ishida, 2000. The research on open source got a boost when the MIT Open-source repository was launched in the summer of 2000, distributing working papers such as Lerner & Tirole, 2000; Lakhani & Von Hippel, 2000; Tuomi, 2000; and Weber, 2000. Many of the early papers were inspired by the descriptions of open-source development models in Raymond, 1998 and DiBona, Ockman, & Stone, 1999.

2 A good early, policy-oriented study was Peeling & Satchell, 2001.

3 Some open-source systems, such as MySQL, use dual licensing, which requires license fees for commercial use.

4 Some empirical and consultant studies, however, lend support to the claim that at least some parts of the Linux system have smaller error ratios than closed-source code. Some survey-based studies have also shown that the speed at which errors are corrected may be faster in open-source than in closed-source projects. In general, comparative research between open- and closed-source projects has so far been rare, however.

5 The distribution of effort among developers has been studied, for example, by Mockus, Fielding, & Herbsleb, 2002, who found that only a few programmers contribute almost all code. The size distribution of open-source projects has been studied by Krishnamurthy, 2002, who found that half of the hundred projects studied had fewer than four developers, with an average of 6.7 developers.

6 Using data from González-Barahona, Ortuño Pérez, et al., 2002, one may estimate that during the first decade of Linux development the effort that went into developing the system was roughly equal to 500 person-years of commercial development. It is difficult to translate this number into actual work hours or developer-community size, as it is difficult to estimate the productivity of Linux-kernel developers without further study. In general, individual programmer productivity differences are often found to vary more than an order of magnitude, and one may assume that, on average, the core Linux programmers have worked with relatively good programming productivity. One might also expect that in recent years the programming effort has increased, partly because many commercial firms are now involved in Linux development.

7 The cultural-historical school was developed by Lev Vygotsky and his colleagues in the 1920s in the Soviet Union. For a historical review of the Vygotskian school and its central ideas, see e.g. Kozulin, 1990 and Wertsch, 1991.

8 In the first half of 2004, an average of 48 new vulnerabilities a week were reported for Windows-based PCs. The total number of reported security vulnerabilities for Windows software reached 10,000 by mid-2004 according to Symantec's Internet Security Threat Report.

9 Historically, microprocessors have included undocumented microcode instructions, for example for testing and because the processor developers have left some options open for the final specification of the processor.

10 Strictly speaking, it is possible that the value of an open-source system may decrease as more people use it. As long as only a few users have found the system, it may have some temporary scarcity value, for example because the early users may benefit from cost advantages that have not yet been appropriated by competitors. An innovative open-source user could even maintain such a competitive advantage by continually adopting state-of-the-art open-source systems. Such situations, of course, can not be described using economic theories that start out from the assumption that economic players operate in an equilibrium.

11 The report, released by the Alexis de Tocqueville Institution in May 2004, was widely discredited by the people quoted in it, but it created an avalanche of commentary in the public press and on the Internet. In Europe, the report's author interviewed Andrew Tanenbaum, the creator of the Minix system, and the

author of the present chapter. After extensive discussions about Microsoft's role in the production and funding of the report, Microsoft eventually repudiated it, commenting that it was an unhelpful distraction.

References

BEZROUKOV, N. (1999). "A second look at the Cathedral and the Bazaar." *First Monday,* 4 *(12)* http:// firstmonday.org/issues/issue4_12/bezroukov/

BROWN, J.S., & Duguid, P. (1991). "Organizational learning and communities of practice: toward a unified view of working, learning, and innovation." *Organization Science,* 2, pp. 40-57.

DALLE, J.M., & JULLIEN, N. (2001). "Open-source vs. proprietary software." Guest lecture at ESSID Summer School, Cargèse.

DEMPSEY, B.J., WEISS, D., JONES, P., & GREENBERG, J. (1999). "A quantitative profile of a community of open-source Linux developers." School of Information and Library Science, University of North Carolina at Chapel Hill. Technical Report TR-1999-05.

DIBONA, C., S. OCKMAN, & STONE, M. (1999). *Open Sources: Voices from the Open-Source Revolution.* Sebastopol, CA: O'Reilly & Associates, Inc.

EDWARDS, K. (2000). "When beggars become choosers." *First Monday,* 5 *(10)* http://firstmonday.org/issues/issue5_10/edwards/index.html.

FANO, R. M. (1967). "The computer utility and the community." IEEE International Convention Record, 30-34. Excerpts reprinted in *IEEE Annals of the History of Computing*, 14 (2), pp. 39-41.

FELLER, J. and FITZGERALD, B. (2000). "A framework analysis of the open-source software development paradigm." The 21st International Conference in Information Systems (ICIS 2000), pp. 55-69.

GHOSH, R.A. (1998). "Cooking-pot markets: an economic model for the trade in free goods and services on the internet." *First Monday,* 3 *(3)* http://firstmonday.org/issues/issue3_3/ghosh/index.html.

GHOSH, R.A., & PRAKASH, V.V. (2000). "The Orbiten Free-Software Survey." *First Monday,* 5 *(7)* http://firstmonday.org/issues/issue5_7/ghosh/index.html.

GONZÁLEZ-BARAHONA, J.M., ORTUÑO PÉREZ, M.A., HERAS QUIRÓS, P., CENTENO-GONZÁLEZ, J., & MATELLÁN-OLIVERA, V. (2002). "Counting potatoes: the size of Debian 2.2." http://people.debian.org/~jgb/debian-counting/counting-potatoes/.

JOHNSON, J.P. (2001). "Economics of Open Source." Available at http://opensource.mit.edu/papers/johnsonopensource.pdf.

KOCH, S., & SCHNEIDER, G. (2000). "Results from software engineering research into open-source development projects using public data." *Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, p. 22.

KOLLOCK, P. (1999). "The economies of online cooperation: gifts and public goods in cyberspace." In M.A. SMITH & P. KOLLOCK (eds), *Communities in Cyberspace*. London: Routledge, pp. 220-239.

KOZULIN, A. (1990). *Vygotsky's Psychology: A Biography of Ideas.* Cambridge, MA: Harvard University Press.

KRISHNAMURTHY, S. (2002). "Cave or Community?: An Empirical Examination of 100 Mature Open-Source Projects." *First Monday,* 7 *(6)* http://firstmonday.org/issues/issue7_6/krishnamurthy/index.html.

Kuwabara, K. (2000). "Linux: a bazaar at the edge of chaos." *First Monday,* 5 *(3)* http:// firstmonday.org/issues/issue5_3/kuwabara/.

Lakhani, K., & Von Hippel, E. (2000). "How open-source software works: 'Free' user-to-user assistance." MIT Sloan School of Management Working Paper, 4117.

Lave, J., & Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation.* Cambridge: Cambridge University Press.

Lerner, J., & Tirole, J. (2000). "The simple economics of open source." National Bureau of Economic Research. NBER Working Paper No. 7600.

Ljungberg, J. (2000). "Open-source movements as a model for organizing." *European Journal of Information Systems,* 9 *(4),* pp. 208-216.

Mockus, A., Fielding, R., and Herbsleb, J. (2000). "A case-study of open-source software development: the Apache server." *Proceedings of the 22nd International Conference on Software Engineering*, pp. 263-272.

Mockus, A., Fielding, R., & Herbsleb, J. (2002). "Two case-studies on open-source software development: Apache and Mozilla." *ACM Transactions on Software Engineering and Methodology,* 11 *(3),* pp. 309-46.

Moon, J.Y., & Sproull, L. (2000). "Essence of distributed work: the case of the Linux kernel." *First Monday,* 5 *(11)* http://firstmonday.org/issues/issue5_11/moon/index.html.

O'Mahony, S. (2003). "Guarding the commons: how community-managed software projects protect their work." *Research Policy,* 32 *(7),* pp. 1179-98.

Peeling, N., & Satchell, J. (2001). "Analysis of the Impact of Open-Source Software." QinetiQ Ltd. QINETIQ/KI/SEB/CR010223. Available at http://www.govtalk.gov.uk/interoperability/egif_document.asp?docnum=430.

Raymond, E.S. (1998). "The Cathedral and the Bazaar." *First Monday,* 3 *(3)* http:// firstmonday.org/issues/issue3_3/raymond/index.html.

Tuomi, I. (1999). *Corporate Knowledge: Theory and Practice of Intelligent Organizations.* Helsinki: Metaxis.

Tuomi, I. (2000). "Learning from Linux: Internet, Innovation and the New Economy. Part 1: Empirical and Descriptive Analysis of the Open-Source Model." SITRA Working Paper, Berkeley, 15 April 2000.

Tuomi, I. (2001). "Internet, innovation, and open source: actors in the network." *First Monday,* 6 *(1)* http://firstmonday.org/issues/issue6_1/tuomi/index.html.

Tuomi, I. (2002). *Networks of Innovation: Change and Meaning in the Age of the Internet.* Oxford: Oxford University Press.

Tuomi, I. (2003) "Open source for human development." Paper presented at the EU-US Workshop on Advancing the Research Agenda on Free/Open-Source Software. Brussels, 14 October 2002. Available at http://www.infonomics.nl/FLOSS/workshop/papers/tuomi.htm.

Tuomi, I. (2004). "Evolution of the Linux Credits file: methodological challenges and reference data for open-source research." *First Monday,* 9 *(6)* http://firstmonday.org/issues/issue9_6/tuomi/index.html

Von Hippel, E., & von Krogh, G. (2003). "Open-source software and the 'private-collective' innovation model: issues for organization science." *Organization Science,* 14 *(2)*, pp. 209-223.

WEBER, S. (2000). “The political economy of open-source software.” BRIE Working Paper 140. http://brie.berkeley.edu/~briewww/pubs/wp/wp140.pdf.

WERTSCH, J.V. (1991). *Voices of the Mind: A Sociocultural Approach to Mediated Action.* Cambridge, MA: Harvard University Press.

YAMAUCHI, Y., YOKOZAWA, M., SHINOHARA, T., AND ISHIDA, T. (2000). “Collaboration with lean media: how open-source software succeeds.” CSCW ‘00, Philadelphia: ACM. pp. 329-338.

Biography

Ilkka Tuomi is currently working with the European Commission's Joint Research Centre, Institute for Prospective Technological Studies, Seville, Spain. He graduated in theoretical physics at the University of Helsinki, and has a PhD on Adult Education from the same university. His recent research has focused on innovation, open source, information society technologies, and the knowledge society. Before joining the IPTS in 2002 he was a visiting scholar at the University of California, Berkeley, where he conducted research on the new dynamics of innovation networks, working with Manuel Castells. From 1987 to 2001 he worked at the Nokia Research Center, most recently as Principal Scientist, Information Society and Knowledge Management. His most recent book, Networks of Innovation (Oxford University Press, 2002), studies Internet-related innovations and the history of computer networking, the World Wide Web, and Linux.