STEVEN WEBER

The Success of Open Source

HARVARD UNIVERSITY PRESS
Cambridge, Massachusetts, and London, England
2004

CHAPTER 3

What Is Open Source and How Does It Work?

In January 1991 a computer science graduate student at the University of Helsinki named Linus Torvalds bought himself a personal computer with an Intel 80386 processor, 4 megabytes of memory, and a 40-megabyte hard drive—quaint in today's computing environment, but quite a powerful personal setup for 1991. Like most PCs at the time, the machine came with Microsoft DOS (disk operating system) as its standard software. Torvalds had no love for DOS. He strongly preferred the technical approach of the UNIX-style operating systems that he was learning about in school. But he did not like waiting on long lines for access to a limited number of university machines that ran Unix for student use. And it simply wasn't practical to run a commercial version of Unix on his PC—the available software was too expensive and also too complicated for the hardware.

In late 1990 Torvalds had heard about Minix, a simplified Unix clone that Professor Andrew Tanenbaum at Vrije University in Amsterdam had written as a teaching tool. Minix ran on PCs, and the source code was available on floppy disks for less than $100. Torvalds installed this system on his PC. He soon went to work building the kernel of his own Unix-like operating system, using Minix as the scaffolding. In autumn 1991, Torvalds let go of the Minix scaffold and released the source code for the kernel of his new operating system, which he called Linux, onto an Internet newsgroup, along with the following note:

> I'm working on a free version of a Minix look-alike for AT-386 computers. It has finally reached the stage where it's even usable (though it

> may not be, depending on what you want), and I am willing to put out the sources for wider distribution. . . . This is a program for hackers by a hacker. I've enjoyed doing it, and somebody might enjoy looking at it and even modifying it for their own needs. It is still small enough to understand, use and modify, and I'm looking forward to any comments you might have. I'm also interested in hearing from anybody who has written any of the utilities/library functions for Minix. If your efforts are freely distributable (under copyright or even public domain) I'd like to hear from you so I can add them to the system.[1]

The response was extraordinary (and according to Torvalds, mostly unexpected). By the end of the year, nearly 100 people worldwide had joined the newsgroup. Many of these contributed bug fixes, code improvements, and new features to Torvalds's project. Through 1992 and 1993, the community of developers grew at a gradual pace—even as it became generally accepted wisdom within the broader software community that the era of Unix-based operating systems was coming to an end in the wake of Microsoft's increasingly dominant position.[2] In 1994, Torvalds released the first official Linux, version 1.0. The pace of development accelerated through the 1990s.

By the end of the decade, Linux was a major technological and market phenomenon. A hugely complex and sophisticated operating system had been built out of the voluntary contributions of thousands of developers spread around the world. By the middle of 2000 Linux ran more than a third of the servers that make up the web. It was making substantial inroads into other segments of computing, all the way from major enterprise-level systems (in banks, insurance companies, and major database operations) to embedded software in smart chips and appliances. And in 1999 Linux became a public relations phenomenon. VA Linux and Red Hat Software—two companies that package and service versions of Linux as well as other open source programs—startled Wall Street when they emerged among the most successful initial public offerings on NASDAQ. Suddenly the arcane subjects of operating systems and source code had moved from the technical journals to the front page of *The New York Times*. And open source became a kind of modern day Rorschach test for the Internet-enabled society.

Chapter 4 contains a detailed history of how open source evolved from about 1990 to the present. This chapter describes the phenomenon: What is open source and how does it function? To make sense of

the data that captures what we know about the open source movement and the people who contribute to it requires an understanding of what we are measuring and why. That sounds obvious, but putting this principle into practice is not so simple. Linux is just one example of an extremely diverse phenomenon. To approach this analytic problem, I use a two-pronged strategy. First, I present a simple and sparse ideal-typical description of an open source project. As an ideal type it captures the major shared characteristics of open source, although it is not itself "true" for any single project.[3]

Second, I situate this ideal type within the framework of a *production process*, a conceptual move central to the logic of this book. The essence of open source is not the software. It is the *process* by which software is created. Think of the software itself as an artifact of the production process. And artifacts often are not the appropriate focus of a broader explanation. If I were writing this book in 1925 and the title was *The Secret of Ford*, I would focus on the factory assembly line and the organization of production around it, not about the cars Ford produced. Production processes, or ways of making things, are of far more importance than the artifacts produced because they spread more broadly. Toyota, for example, pioneered lean production in a factory that made cars. Twenty years later, this way of making things had spread throughout the industrial economy. Similarly, open source has proved itself as a way of making software. The question becomes, what are the conditions or boundaries for extending open source to new kinds of production, of knowledge and perhaps of physical (industrial) goods as well? Intriguing questions—but not answerable until we have a more sophisticated understanding of what the open source production process is, how it works, and why.

This chapter describes the open source process by situating it within the "problem" that it is trying to "solve" and then focusing on the people who contribute to open source software and how they relate one to another. I pose and answer, as far as possible given the limitations of the data and the variation among different open source projects, four ideal-type questions.

- Who are the people who write open source code?
- What do these people do, exactly?
- How do they collaborate with each other?
- How do they resolve disagreements and deal with conflict?

This sets the stage for explaining the deeper puzzles of the open source process in Chapter 5.

The Software "Problem"

To build complex software is a difficult and exacting task. The classic description of what this feels like comes from Frederick Brooks, who likened large-scale software engineering to a prehistoric tar pit:

> One sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks. Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it . . . Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion.[4]

In 1986 Brooks chaired a Defense Science Study Board project on military software. Afterward he wrote a paper entitled "No Silver Bullet: Essence and Accidents of Software Engineering."[5] This paper, while controversial, still stands as the most eloquent statement of the underlying structure of the software engineering problem—and why it is so hard to improve. Brooks uses Aristotelian language to separate two kinds of problems in software engineering. *Essence* is the difficulty inherent in the structure of the problem. *Accident* includes difficulties that in any particular setting go along with the production of software, or mistakes that happen but are not inherent to the nature of the task.

Brooks's key argument is that the fundamental challenge of software lies in the essence, not in the accidents. The essence is the conceptual work of building the interlocking concepts that lie behind any particular implementation—data sets, relationships among data, the algorithms, the invocations of functions. To implement this essence by writing working code is hard, to be sure. But those kinds of practical coding difficulties, for Brooks, fall into the realm of accident. Accidents can be fixed or at least made less common by evolving the process. But software will remain hard to write because "the complexity of software is an essential property not an accidental one."[6]

If this is correct, simple models fail because the complexities at the

core of the task cannot be abstracted away. A physicist dealing with complexity has the advantage of being able to assume that models can be built around unifying physical principles. The software engineer cannot make that assumption. Einstein said that there must be simplifiable explanations of nature because God is not arbitrary. But there is no such stricture for software engineering because the complexity at play is "arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which [the programmer's] interfaces must conform."[7]

To make matters worse, humans use software in an extraordinarily diverse technological and cultural matrix that changes almost continuously. If an auto engineer has to envision the range of conditions under which people will drive a car, the software engineer is faced with a harder task, if for no other reason than that much of the technological environment in which a piece of software will be used has not even been invented at the moment that the software is being written. Highways and bridges, in contrast, don't change that fast, and they are not configurable by users in the way that software is.

Another aspect of this complexity is that software is invisible and, more importantly, "unvisualizable." Brooks means that software is hard or perhaps even impossible to represent in a spatial or geographical model. Silicon chips have wiring diagrams that are incredibly intricate but at least they exist on one plane. Software structure exists on multiple logical planes, superimposed on one another. Software is conceptually more like a complex poem or great novel in which different kinds of flows coexist across different dimensions. To represent any one of these flows is possible. You can diagram the syntax of a poem or write an essay about an underlying theme. To represent all at once—and to do so in a way that communicates effectively to an outside observer—is a problem of a different order of magnitude, perhaps insoluble.

That is why great poetry is almost always the product of a single creative mind. It can be helped along, of course. Design practices and general rules can be and are taught to aspiring poets, and to aspiring software designers. Technology provides both with tools to assist their work, from word processors to elegant test programs for software modules. But technology cannot now, and will not in the foreseeable future, solve the problem of creativity and innovation in nondecomposable complex systems. The essence of software design, like the writing of poetry, is a creative process. The role of technology and organization is to liberate that creativity to the greatest extent possible and to facilitate its translation into working code. Neither new technology nor a "better" division of labor can replace the creative essence that drives the project.

Hierarchical and Open Source "Solutions" as Ideal Types

There is more than one way to skin this cat.[8] The fairy tale solution would be to place a brilliant young eccentric in an isolated basement room with a computer and lots of coffee and let her write software until the point of exhaustion. In fact a great deal of software does get written in exactly this way. But most of this software is used only by the author or perhaps a few friends. And there are inherent limits to software that can be built by one or two people. One person can write a utility, a device driver, or some other small program in a matter of days or weeks. A modern operating system, however, consists of millions of lines of code. And scale is not the only issue. Like a modern car, with its engine, brakes, electronics, hydraulics, and so on, software is made up of components that call on very different kinds of expertise. Yet the result must be conceptually coherent to the single mind of the user.

One way or another, the software problem leads inexorably to some kind of division of labor. Putting large numbers of people into the correct slots in a division of labor is important. But getting the numbers of people right and putting them in the right places is really a secondary problem. The primary question is, *What kind of division of labor, organized how?*

In 1971 Harlan Mills put forward an evocative image in response to this question. It was obvious to him that a large software project must be broken up so separate teams can manage discrete pieces. The key to Mills's argument was that each team should be organized as a surgical team, not a hog-butchering team. In other words, "instead of each team member cutting away on the problem, one does the cutting and the others give him every support that will enhance his effectiveness and productivity."[9]

Frederick Brooks took this argument a step further with an analogy to the building of medieval cathedrals. But Brooks meant a particular

kind of medieval cathedral. He was talking about Reims, not Chartres. In fact most European cathedrals are a mishmash of architectural designs and styles, built at different times according to the aesthetic of their designers. Norman transepts may abut a Gothic nave. These contradictions produce a certain kind of splendor in a cathedral, because the human eye can move with ease across boundaries and find beauty in the dissonance. Data cannot do this, which is why similar design contradictions are a nightmare in software.

The key to software design, for Brooks, is conceptual integrity, the equivalent of architectural unity that comes from a master plan. His argument about the appropriate division of labor follows directly from this commitment. Conceptual integrity "dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds."[10] Only a single great mind can produce the design for a great cathedral. The division of labor for coding (in other words, building the cathedral) then proceeds along two clear lines.

First, draw a separation as cleanly as possible between architecture and implementation. The architect designs the system, creates the master plan, and owns the mental model of the intended outcome. The architect is also responsible for dividing the system into subsystems, each of which can be implemented as independently as possible. Second, structure implementation teams like surgical teams, as Mills argued. Each surgical team has its own subarchitect who is responsible for organizing the implementation team that works under him (just as a chief surgeon assigns tasks in the operating room). The process, in principle, can advance recursively into a multilayered division of labor, depending on the complexity of the project that the master architect is trying to construct.

Stripped to its core, the Brooks approach is really a slightly modified Fordist style of industrial organization. That is no criticism: Fordist divisions of labor are incredibly successful at building certain kinds of products. A clear division between architecture and implementation, segmentation of tasks into subsystems that are then supposed to "snap" together, reporting hierarchies with command and control from above, are all familiar techniques of industrial organization. And they all fit well within a traditional sketch of an ideal-typical corporate hierarchy. An authority assigns tasks, monitors for performance, and compensates according to measurable indicators of execution.

This is not nearly a perfect solution, even in theory. The dilemmas are familiar. Monitoring and evaluating the performance of a complex task like writing code is expensive and imperfect. Proxy measures of achievement are hard to come by. Quality is as important (often more important) than quantity, and simple measures are as likely to be misleading as informative (someone who produces a large number of lines of code may be demonstrating poor implementation skills, not productivity). Shirking within teams and free riding on the efforts of others is hard to isolate. One person's good efforts can be rendered ineffective by another person's failure to produce.

Much of the software engineering and organization literature focuses on ways to ameliorate at least some of these problems in practice. The underlying notion is just like Winston Churchill's views about democracy: Building software this way is the worst possible system except for all the others. Improve the implementation (by removing what Brooks called "accident") over time and you move gradually toward a better industrial organization for software. Substantial progress has in fact been made in exactly this way.

But the essence of the problem according to Brooks—the conceptual complexity of design—will remain. This argument is now commonly called Brooks's Law and it is foundational in programming lore. The simple version of Brooks's Law is this: Adding more manpower to a software project that is late (behind schedule) will make the project even later. Hence the phrase "the mythical man-month."

What lies behind the mythical man-month is a subtle line of reasoning about the relationship between complex systems of meaning and the imperfections of human communication. Brooks says that, as the number of programmers working on a project rises (to n), the work that gets done scales at best by n—but vulnerability to bugs scales as the square of n. In other words, the production system tends to create problems at a faster rate than it creates solutions.

Too many cooks spoil the broth is an old argument. What Brooks's Law adds is a statement about *the rate at which that happens.* Why does vulnerability to bugs scale as the square of n? Brooks argues that the square of n represents a decent estimate of the number of potential communications paths and code interfaces between developers, and between developers' code bases. Human communication about complex, often tacit goals and objectives is imperfect and gets more imper-

fect, *and at an increasing rate*, as it must travel among larger numbers of people. The complexity of technological interfaces between code modules increases in similar geometric fashion. This is the essential problem of software engineering. Removing Aristotelian accidents only reduces the rate at which the underlying problem gets worse. Indeed, as software systems evolve toward greater complexity, organizations will be challenged to keep up, running faster to stay in the same place.

The open source process takes on this challenge from a different direction. The popular image of open source as a bazaar does capture the feeling of an ideal type. It is an evocative image. But it is analytically misleading and it is best to drop it. *The key element of the open source process, as an ideal type, is voluntary participation and voluntary selection of tasks.* Anyone can join an open source project, and anyone can leave at any time. That is not just a free market in labor. What makes it different from the theoretical option of exit from a corporate organization is this: Each person is free to choose what he wishes to work on or to contribute. There is no consciously organized or enforced division of labor. In fact the underlying notion of a division of labor doesn't fit the open source process at all. Labor is *distributed*, certainly—it could hardly be otherwise in projects that involve large numbers of contributors. But it is not really divided in the industrial sense of that term.

The ideal-type open source process revolves around a core code base. The source code for that core is available freely. Anyone can obtain it, usually over the Internet. And anyone can modify the code, freely, for his or her own use. From this point the process differs among projects, depending largely on how they are licensed. BSD-style licenses are minimally constraining. Anyone can do almost anything with this code, including creating from it a proprietary product that ships without source code. The GPL is much more constraining. In essence, anyone is free to do anything with GPL code *except things that restrict the freedom of others to enjoy the same freedoms.* In practice this means that a program derived from GPL code must also be released under the GPL with its source code.

The key to the open source process is only partly what individuals do for themselves with the source code. It is also in what and how individuals contribute back to the collective project. Again there are differences. BSD-style projects typically rest with a small team of developers

who together write almost all the code for a project. Outside users may modify the source code for their own purposes. They often report bugs to the core team and sometimes suggest new features or approaches to problems that might be helpful. But the core development team does not generally rely heavily on code that is written by users. There is nothing to stop an outside user from submitting code to the core team; but in most BSD-style projects, there is no regularized process for doing that. The BSD model is open source because the source code is free. But as an ideal type, it is not vitally collaborative on a very large scale, in the sense that Linux is.

The vital element of the Linux-style process is that the general user base can and does propose what are called "check-ins" to the code. These are not just bug reports or suggestions, but real code modifications, bug fixes, new features, and so on. The process actively encourages extensions and improvements to Linux by a potentially huge number of developers (any and all users). If there is a general principle of organization here, it is to lower the barriers to entry for individuals to join the debugging and development process. As an ideal type, the Linux process makes no meaningful distinction between users and developers. This takes shape in part through a common debugging methodology that is derived from the Free Software Foundation's GNU tools. It takes shape in part through impulsive debugging by someone trying to fix a little problem that she comes across while using the software. And it takes shape in part through individuals who decide to make debugging and developing Linux a hobby or even a vocation.

But the process of developing and extending Linux is not an anarchic bazaar. The email discussion lists through which users share ideas and talk about what they like and don't like, what works and what doesn't, what should be done next and shouldn't (as well as just about everything else) do have a raucous, chaotic feel to them. Conflict is common, even customary in these settings. Language gets heated. There are indeed norms for the conduct of these discussions that bound what kinds of behaviors are considered legitimate. The principal norm is to say what you think and not be shy about disagreeing with what others, including Linus Torvalds, might think. Yet the procedure for reviewing submissions of code and deciding whether a submission gets incorporated into the core code base of Linux is ordered

and methodical. A user-programmer who submits a patch for inclusion in Linux is expected to follow a procedure of testing and evaluation on his own, and with a small number of colleagues, before submitting the patch for review. The submission then travels up through a hierarchy of gatekeepers or maintainers who are responsible for a particular part of the code base, lieutenants who oversee larger sections of code, and eventually Linus Torvalds, who de facto makes the final decision for all official code modifications.

This hierarchy has evolved and grown more elaborate over time as Linux itself has grown. Smaller open source projects have simpler and often more informal decision-making systems. Apache, on the other hand, has a formal de facto constitution that is built around a committee with explicit voting rules for approval of new code. The big question is, Why are these systems stable? Why do people obey the rules and accept decisions that go against their own work?

In fact, sometimes they don't. And in an open source setting there is no reason why they must. An individual whose code patch gets rejected always has a clear alternative path. He can take the core code, incorporate the patch, set the package up as a "new" open source project, and invite others to leave the main project and join his. This is called "forking the code base" or simply "forking." Open source guarantees this right—in fact, some developers believe that the essential freedom of free software is precisely the right to fork the code at any time.

In practice, major forks are rare. In practice, most participants in open source projects follow the rules most of the time. There is a lot to explain here. The point of this discussion is simply to set the context for that explanatory challenge.

Decentralized voluntary cooperation is always an interesting phenomenon in human affairs. For some social scientists, it is almost foundational. For studies of how the Internet may change political economy and society by enabling new kinds of communities and other cooperative institutions, it is crucial. The problem certainly gets more interesting when it involves highly motivated and strongly driven individuals who clearly have attractive options to exit any particular cooperative arrangement.

Brooks's Law adds a particularly challenging dimension to the problem as it manifests in software development. To explain the open source process, we need a compelling story about why individuals contribute time and effort to write code that they do not copyright and for which they will not be directly compensated for a collective project whose benefits are nonexcludable. In other words, any individual can take from the project without contributing in return.

But explaining individual motivations does not explain the success of open source. In a peculiar way, it makes the problem of explanation harder. *If Brooks is even partially right about the nature of complexity, then the success of open source cannot simply depend on getting more people or even the "right" people to contribute to the project. It depends also, and crucially, on how those people are organized.*

The reason a great poem is written by a single person and not by thousands of contributors from all over the world is not that it would be hard to get those thousands of people to contribute words to the collective poem, but that those words would not add up to anything meaningful. They would simply be a mess of uncoordinated words that no one would see as a poem (certainly not a great poem). Eric Raymond famously said about the open source development process, "with enough eyeballs all bugs are shallow."[11] Whether he is right depends on how those eyeballs are organized.

What do we know, descriptively, about the important parameters to help answer these questions?

Who Participates in the Open Source Process?

I would like to start with a clean number that decently estimates how many people participate in open source development. It's not possible to do that, and the problem is not just about measurement. It's about conceptualization: Should we define as an open source developer a high school student who modifies some source code for her unusual configuration at home, or reports a bug to one of thousands of small open source projects listed on the website SourceForge.net? Should we limit the definition to people who contribute a certain threshold number of lines of code to a major project like Linux or Apache? Rather than try to define *a priori* the conceptual boundaries, I think it is better for now to remain agnostic and look broadly at what kinds of data are available, to give a more textured view of the size and characteristics of an (evolving and dynamic) open source community.

SourceForge is a major website for open source development proj-

ects that provides a set of tools to developers. It is also a virtual hangout, a place that open source developers visit regularly to see what kinds of projects are evolving and who is doing what in specific areas.[12] In July 2001 SourceForge reported 23,300 discrete projects and 208,141 registered users; in September 2003 there were 67,400 projects and over 600,000 registered users. Most of these projects are very small, both in technical scope and in the number of people working on them. Some are essentially dead in the water or abandoned. With these caveats, the numbers are suggestive of the scope of activity in at least one very active part of the open source community.

Counter.li.org is an effort to count the number of active Linux users over time. It relies on voluntary registration for one bottom line measure, but also tries to estimate the size of the community by a variety of techniques that vary in sophistication and plausibility. The range of estimates is huge, with a consensus guesstimate of about 18 million as of May 2003.[13] This roughly tracks estimates made by Red Hat Linux, the major commercial supplier of packaged versions of the Linux operating system (and thus the company most highly motivated to generate a serious assessment of market size). Even if this number is right in some very broad sense, it says nothing about the scope of contributions. Only a subset of users contributes in significant ways to the development of Linux.

There are several large research efforts, both completed and ongoing, aimed at collecting more precise statistics about active contributions and contributors to open source software development.[14] Probably the most ambitious effort is the Orbiten Free Software Survey, carried out over eighteen months in 1999 and 2000 by Rishab Ghosh and Vipul Ved Prakash.[15] Prakash wrote a software tool called CODD that tries to automate the process of identifying credits in source code.[16] He and Ghosh ran this tool across an important but still quite limited subset of open source projects.[17] Within this subset, making up about 25 million lines of code, they found 3,149 discrete open source software projects and 12,706 identifiable developers. Another 790 developers were unidentifiable within the data.

Many other efforts to collect raw data on developers focus specifically on Linux. These studies show that the community of developers contributing to Linux is geographically far flung, extremely large, and notably international. It has been so nearly from the start of the

project. The first official "credits file" (which lists the major contributors to the project) was released with Linux version 1.0 in March 1994. This file included seventy-eight individual developers from twelve countries and two developers whose home countries were not disclosed. Ilka Tuomi adjusted these numbers to take account of the different sizes of home countries to show the disproportionate influence of Europeans and the relatively small contribution of developers living in the United States (Figure 1).[18]

Of the major developers listed in the credits file for the 2.3.51 release (March 2000), the United States had the largest absolute number, but Finland was still by far the most active on a per capita basis. Thirty-one countries were represented. Clearly, the international aspect of Linux development has not decreased over time or with the increasing notoriety of the software (Figure 2).

Gwendolyn Lee and Robert Cole looked at the institutional affiliation of contributors to the Linux kernel from the 2.2.14 credits file.[19] The top-level domain of a contributor's email address (such as .org,

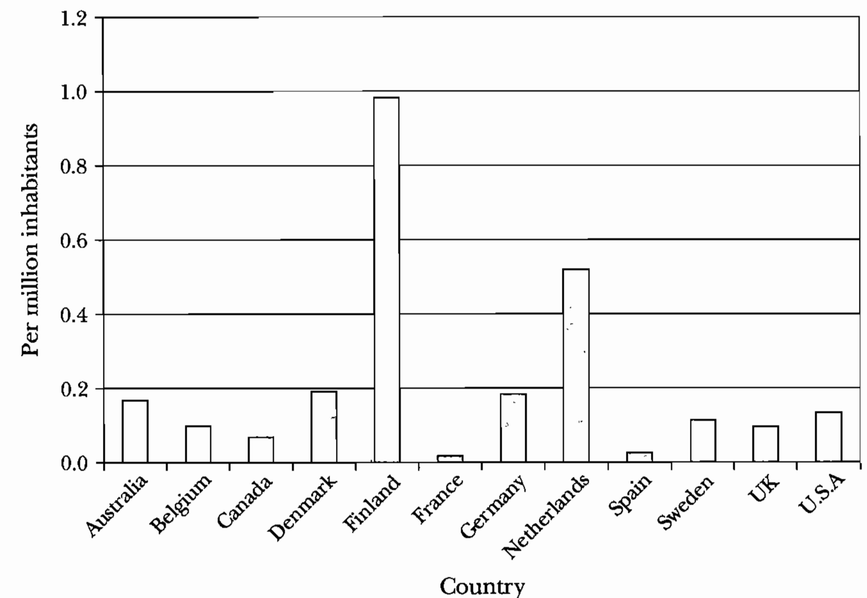

Figure 1 Linux code authors listed in first credits file (1994), concentration by country.

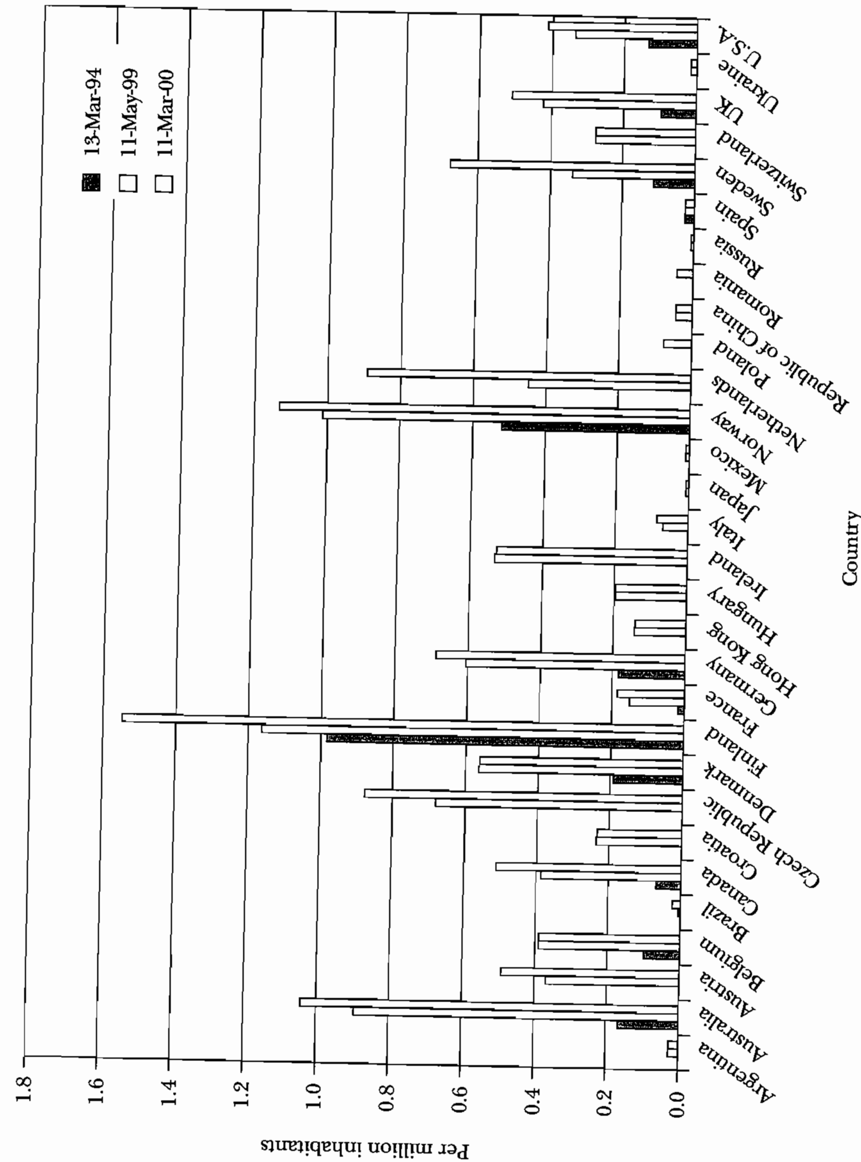

Figure 2 Linux code authors listed in credits files, concentration over time.

.com, or .edu) is an imperfect measure of institutional affiliation, but it is a reasonable proxy.[20] Their observation that more developers have .com than .edu addresses at least calls into question another common perception. Academics and computer science students (who presumably write code for research and teaching) may not dominate the open source process (see Figure 3).

But these numbers count only the major contributors to the Linux kernel. Other active developers report and patch bugs, write small utilities and other applications, and contribute in less elaborate but still important ways to the project. The credit for these kinds of contributions is given in change logs and source code comments, far too many to read and count in a serious way. It is a reasonable guess that there are at least several thousand, and probably in the tens of thousands, of developers who make these smaller contributions to Linux.[21]

A 1999 assessment of these so-called application contributors used Linux software maps (LSMs) located at University of North Carolina's Metalab, one of the oldest and most comprehensive repository sites for Linux software.[22] LSMs are small descriptive metadata files that

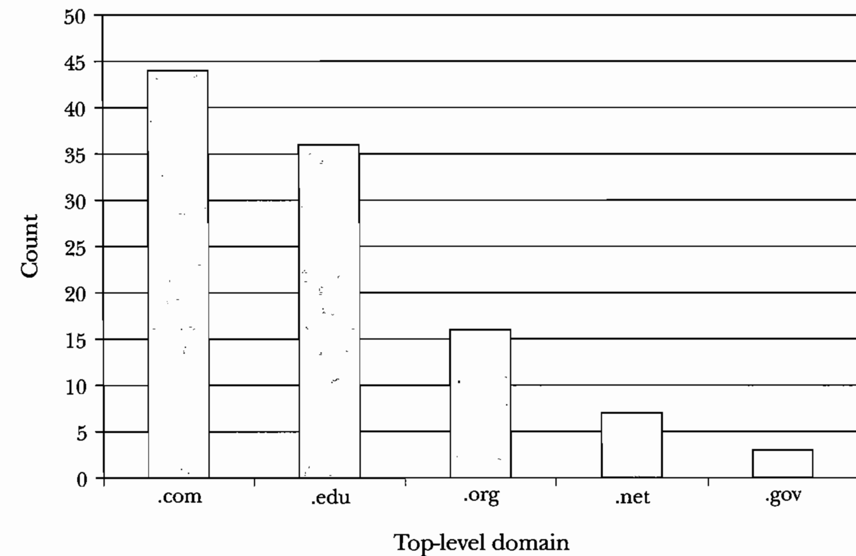

Figure 3 Linux code authors in 2.2.14 credits file, by top-level domain of email address.

many (but not all) developers use to announce their new software and describe its functions.[23] In September 1999 the Metalab archives contained 4,633 LSMs. The distribution among top-level domains reinforces the results of the kernel surveys about geographic distribution of contributors, but shows an even more pronounced European influence (see Figure 4).

In fact 37 percent of the LSMs have email suffixes representing European countries (.de for Germany, .fr for France, and so on) compared to the 23 percent that have .com suffixes. And this method undercounts European participation because at least some authors with .com and other addresses would be located in Europe.

Each of these studies attempts to measure the relative concentration of contributions. In any collective project, not all contributors work equally hard or make contributions that are equally important, and those two variables are not necessarily correlated. Anyone who has worked or played on a team knows the apocryphal 80–20 rule: 80 percent of the work seems to be done by 20 percent of the people.

Although the studies use measures like lines of code or gross numbers of submissions as (deficient) proxies for the value of an individual's contributions, the open source process is not seemingly much different in this respect from other team efforts. The Orbiten Free Software Survey found that the most prolific 10 percent of developers are credited with about 72 percent of the code; the second decile contributes another 9 percent. In fact the top ten individual authors are responsible for almost 20 percent of the total code. Authors tend to focus their attention tightly: About 90 percent of software developers participate in only one or two discrete projects; a small fraction (about 1.8 percent) contributes to six or more projects. The UNC Metalab data shows a similar distribution of effort: Large numbers of LSM authors (about 91 percent) have contributed only one or two items to the archive, with only a few developers (about 1 percent) contributing more than six.[24] A more recent study on the code repository for the GNOME project (a desktop environment that runs on top of an operating system) finds similar patterns—a relatively small inner circle of developers who contribute a majority of the code and are the most active participants in the email list discussions of the project, surrounded by a larger group of less active coders and discussion group participants.[25]

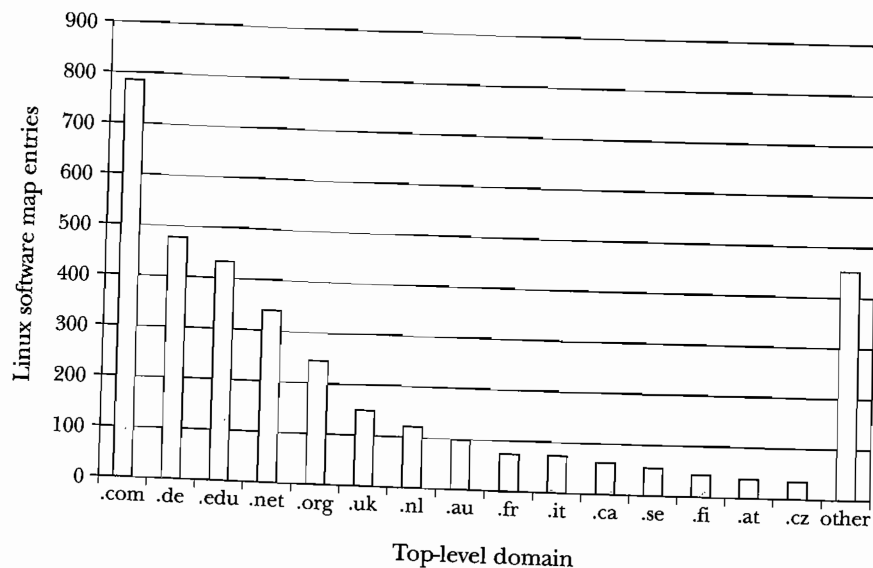

Figure 4 Linux software map (LSM) entries, by top-level domain of email address.

This data certainly tempers the image of a babbling anarchic bazaar. Open source is a distributed production process, but clearly the distribution is lumpy and somewhat top heavy. The data on Linux is consistent with an image of several *hundreds* of central members who do most of the coding, and several *thousands* of comparatively peripheral participants who contribute in more indirect and sporadic fashion. For the most part contributors work on one or a very small number of programs, rather than spread their efforts more widely over a broad range of projects. It is important to remember the caveats. Each survey relies on a limited sample. More important, none of this data speaks directly to the relative importance of any particular contribution or even the level of effort that any individual puts in. One could in a few days write many lines of relatively simple code to solve a shallow bug, or spend weeks working on a small piece of code that solves a very difficult problem and removes a major roadblock from a project. Still, the data does say something about the demography of the open source community, including its profoundly international nature and its superficial (at least) resemblance to many other communities in which the 80–20 rule seems to apply.

The data also suggests tentative hypotheses about *why* particular individuals write open source software. For example, the prevalence of developers with .com addresses may indicate that many people write open source code in the course of their everyday work. This possibility is consistent with the narrative that many open source developers tell about themselves: People within commercial organizations like to use open source software because they can customize it for their needs and configurations. These in-house developers write bug fixes and other patches for their own work and then contribute that code back to the community for others to use in similar settings.

Several surveys have tested this story as well as other claims about what motivates individuals to contribute to the open source process. Again there are serious data problems, not the least of which follow from the uncertain boundaries around the overall population (and subpopulations) of open source developers from whom these surveys sample. The more important caveat is the manner in which surveys about individual motivations are embedded, deeply, in the explanatory problem that the researcher believes should guide the study of open source. Put simply, if you accept the simple collective action story as capturing the problem to be solved, you then frame the question of motivation in terms of what drives individuals to make contributions. You design a survey instrument to ask questions aimed at eliciting motivations that would counter the behavioral driving forces (in other words, egoistic rationality) that are assumed to set up the collective action problem in the first place. The survey findings about motivations (however inexact they may be as data) thus are linked to assumptions about explanation in a profound, conceptual way. As I have said, the collective action problematic is too narrow to capture the most interesting puzzles about open source and thus I do not present that data in this chapter. For this reason I do not want to present survey data about individual motivations in this chapter, which is dedicated to describing the open source process. That survey data is best understood in the context of contending explanatory claims, and I present both in Chapter 5.

What Do They Do?

The conventional language of industrial-era economics identifies producers and consumers, supply and demand. The open source process scrambles these categories. Open source software users are not consumers in the conventional sense. By that I mean more than the simple fact that no one consumes software in the sense that one consumes a loaf of bread because software can be copied an infinite number of times at no cost. I mean, more fundamentally, that users integrate into the production process itself in a profound way.

Not all users do this, of course. You can choose to consume open source software in the same way as you consume a copy of Windows. But with open source, that is a voluntary decision, not one that is made for you in the technology and legal protections around the product. On the contrary, the technology and the licensing provisions around open source positively encourage users to be active participants in the process. Many do so, and at many different levels.

The logic of what open source user-programmers do did not emerge from abstract theory. No one deduced a set of necessary tasks or procedures from a formal argument about how to sustain large-scale, decentralized collaboration. It was a game of trial and error—but a game that was played out by people who had a deep and fine-grained, if implicit, understanding of the texture of the community they were going to mobilize. Over time, observers studied the behavior as it played out in practice and tried to characterize its key features. Drawing heavily on Eric Raymond's keen analysis supplemented by a set of interviews and my own observations, I offer eight general principles that capture the essence of what people do in the open source process.

1. MAKE IT INTERESTING AND MAKE SURE IT HAPPENS. Open source developers choose to participate and they choose which tasks to take on. There is naturally, then, a bias toward tasks that people find interesting, significant and "cool," in the community vernacular. Anyone who has done programming knows that much of the work involved in writing (and particularly in maintaining) code is boring, grunt work.

Open source developers clearly look for cool opportunities to create new and exciting functions or do hard things in an elegant way. They also look for opportunities to learn and to improve skills. These are two different kinds of benefits to the volunteer, and either one (or some combination of both) is balanced against the costs. Given a large base of volunteers with diverse interests and expertise, a project leader can hope that someone "out there" will find a particular task either

cool or valuable as a learning experience relative to the time and energy costs, and choose to take it on. Project leaders often engage in friendly marketing, explaining to the "audience" why it would be a great thing for someone to do x, y, or z. A charismatic leader like Torvalds can go a step further, acknowledging (as he often does) that a particular task is not very interesting but someone really should step up to the plate and do it. There are implied benefits within the community for taking on these kinds of tasks.

Volunteers, regardless of why they choose to volunteer, don't like to see their efforts dissipated. A cool program is really only as cool as others say it is; and for the developer to get that kind of feedback, the program needs to be used. Even if someone takes on a task primarily for the sake of her individual learning, she gains additional satisfaction if the task contributes to something more than her own human capital. These may be side benefits of a sort; but if they were not important, individuals would go to problem sets in textbooks rather than to SourceForge.net.

This sets up an interesting problem for a project leader, who needs to search out a workable balance between exciting challenges and a credible assurance that the challenges will indeed be met. User-programmers look for signals that projects will generate significant products rather than turn into evolutionary dead ends. They want interesting puzzles to solve along the way.

2. SCRATCH AN ITCH. Public radio fund-raisers know intuitively what economists have formally argued: One way to get voluntary contributions is to link the contribution to a private good. Hence KQED offers you a T-shirt in "exchange" for your membership dollars. The open source process knows this as well. Most computer programmers love T-shirts. But what they love even more is a solution to a tangible problem that they face in their immediate environment. This is the "itch" that the developer feels, and scratching it is a reliable source of voluntary behavior.

To understand just how important this is, you need to understand something that outsiders generally do not know about the structure of the software industry. Software code written for sale outside the enterprise that writes it is just the tip of a huge programming iceberg, the rest of which is invisible to most people. Mainstream estimates are that 75 percent of all code is written in-house, for in-house use, and not sold by software vendors.[26] For example, NASA engineers spend a great deal of time writing software to orient the solar panels on the space shuttle. Most software is like this—beneath the surface, in-house code built for financial and database system customizations, technical utilities specific to a particular setting, embedded code for microchip-driven toasters, and so on. And most of what programmers get paid to do is debugging and maintaining this code as the environment in which it is being used evolves.

Scratching an itch is usually about solving one of these immediate and tangible problems in an enterprise setting. Or it might be about a computer hobbyist at home trying to get a particular application to talk to a particular device on that person's uniquely configured hardware. Open source developers scratch these itches, just like all programmers do. What is different about the open source process is that the community has developed a system for tapping into this vast reservoir of work and organizing it in a useful way so at least some of it can be brought back in to benefit collaborative projects. This system blurs the distinction between a private good and a public good and leads directly to the third principle.

3. MINIMIZE HOW MANY TIMES YOU HAVE TO REINVENT THE WHEEL. A good programmer is "lazy like a fox." Because it is so hard and time consuming to write good code, the lazy fox is always searching for efficiencies. Open source developers have a particularly strong incentive to take this approach because they are not compensated directly for their time. The last thing a programmer, particularly a volunteer programmer, wants to do is build from scratch a solution to a problem that someone else has already solved or come close to solving.

Open source developers have two major advantages here. First, the code moves freely across corporate boundaries. A huge repository of code gets bigger as more developers use it and contribute to it. The second advantage is at least as important. Developers know that source code released under an open source license like the GPL will always be available to them. They don't have to worry about creating dependencies on a code supplier that might go bankrupt, disappear down the road, or change the terms of access. They don't have to worry about a single supplier trying to exploit a lock-in by later enclosing code or raising price barriers.

Developers working in proprietary corporate settings often com-

plain about the inefficiencies associated with what they call the NIH (not invented here) syndrome. They perceive corporate hierarchies as irrational in their desire to own or at least build for themselves code that they know exists in other corporate hierarchies. But in the context of proprietary software, the NIH syndrome is not necessarily irrational, because dependence on source code that someone else owns can create exploitable dependencies over a very long term.[27] Open source developers are free of this constraint, what Oliver Williamson calls the "hold-up" problem. As a result, they do not have to reinvent the wheel quite so many times.

4. SOLVE PROBLEMS THROUGH PARALLEL WORK PROCESSES WHENEVER POSSIBLE. Computer scientist Danny Hillis has said, "There are only two ways we know of to make extremely complicated things. One is by engineering, and the other is evolution."[28] A software bug fix or a desired feature in a complex program is often an extremely complicated thing. What do open source developers do to "make" it?

Frederick Brooks's description of the traditional software development process relies on the engineering archetype. The architect sets the definition of the problem that the hierarchy below her is going to solve. She plots out a conceptual course (or perhaps more than one potential course) to a solution and then divides the implementation work among a certain number of carefully selected people. Three decisions are critical. What route (or routes) is most promising to take toward a solution? How many people need to be tasked with that job? And which people?

Open source developers rely on the evolution archetype. In some cases, but not all, a project leader does set the effective definition of the problem that needs to be solved. Regardless, if it is an important problem, it will probably attract many different people or perhaps teams of people to work on it. They will work in many different places at the same time, and hence in parallel. They will experiment with different routes to a resolution. And they will produce a number of potential solutions. At some point a selection mechanism (which is messy and anything but "natural selection") weeds out some paths and concentrates on others. This evolutionary archetype works through voluntary parallel processing. No central authority decides how many paths

ought to be tested, how many people ought to be placed on each path, or which people have the most appropriate skills for which tasks.

Remember that these are archetypes and real-world development processes are not so starkly differentiated. It's also important to remember that both archetypes contain their own inefficiencies. Engineering is great if the chief engineer makes all the right decisions. Parallel problem solving at best is messy, like evolution. It is not possible to say *a priori* which is more efficient and less wasteful for a given setting. The point here is that the open source process enables voluntary parallel processing by as many (or as few) developers as the problem can attract, developers who make their own choices about where and how to allocate their resources.

5. LEVERAGE THE LAW OF LARGE NUMBERS. The key to field testing products such as washing machines or cars is to try out the product in as many different settings as possible. Field testers try to predict all the ways in which people will use the product and then test whether the product works for those applications.

The field-testing problem is orders of magnitude more complicated for software and different in kind than it is for a car—but not only because the hardware on which the code must run changes so much faster than roads and bridges. The issue is that even a moderately complex program has a functionally infinite number of paths through the code. Only some tiny proportion of these paths will be generated by any particular user or field-testing program. Prediction is actually the enemy in software testing. The key is to generate patterns of use that are inherently *unpredictable* by the developers. As Paul Vixie puts it, "The essence of field testing is *lack* of rigor."[29]

The more bugs you generate and the sooner you generate them, the better your chances are that you will fix a decent proportion of what will go wrong once the software is in general use. Hence the benefit of large numbers. Proprietary software companies are constrained in this game. Of course they engage in beta testing, or prerelease of an early version to testers and users who are asked to report back on bugs. But companies face a very tricky calculation: How buggy a piece of software are they willing to put out for public assessment? From a developer's perspective, the earlier the beta is, the better. From a marketer's perspective, buggy software is a nightmare. The cost to a company's

reputation as a builder of reliable software can be prohibitive. Customers expect a finished or almost finished product. (Academics understand this calculation all too well. If I were to distribute early drafts of scholarly articles to a broad community of readers to get a diverse set of critical comments, I would quickly destroy my reputation. Instead I give them to a few close friends; but their base of knowledge and expertise tends to be parallel to mine, so what I get in feedback (my beta test result) is narrow and ultimately less useful.)[30]

The open source process has a distinctly different culture that leverages the law of large numbers and exploits the strength of weak ties. The expectations are different: In a real sense open source software is always in beta. The difference in part is the availability of the source code, which empowers continual modification. There is also a collective perception of the open source software process as ongoing research. Open source developers think of themselves as engaging in a continuing research project in which bugs are challenges (not problems) and puzzles (not weaknesses).

As with other evolutionary processes, large numbers and diversity should accelerate adaptation to the environment—in this case, the identification and the fixing of bugs. Again, "with enough eyeballs all bugs are shallow." What this really means is four things. First, different people doing different things with the software will surface more bugs, and that is good. Second, the bugs will be characterized and that characterization communicated effectively to a large group of possible "fixers." Third, the fix will be relatively obvious to someone. And fourth, that fix can be communicated back and integrated efficiently into the core of the project.

Large numbers require organization to work effectively. This principle becomes more apparent when you take into account the observation, common among software developers, that the person who finds a bug and the person who fixes it are usually not the same person. There are probably interesting psychological reasons why that tends to be true, but the observation is an *a priori* argument for the desirability and relative efficacy of parallel debugging. Of course this approach also vastly increases organizational demands on the software development process.

6. DOCUMENT WHAT YOU DO. Source code is readable, but that does not mean it is easy to read. In a sufficiently complex program,

even excellent code may not always be transparent, in terms of what the writer was trying to achieve and why. Like a complex set of blueprints, good documentation explains what the designer was thinking and how the pieces of the design are supposed to fit together. Good documentation takes time and energy. It is often boring and has almost no immediate rewards. The incentives for a programmer to carefully document code are mainly to help others and to ensure that future developers understand what functions a particular piece of code plays in the larger scheme. In corporate settings detailed documentation tends to carry a low priority relative to more immediate tasks. Much of the communication about code happens in less formal settings in which exchange of tacit knowledge substitutes, at least in the short term, for strict documentation.

Open source developers, in contrast, have to rely more heavily on good documentation. A voluntary decentralized distribution of labor could not work without it. Potential user-programmers, connected in most cases only by bandwidth, need to be able to understand in depth the nature of a task they might choose to take on. Members of this community understand that documentation is a key means of lowering the barriers to entry for user-programmers, particularly those whom they will never meet. An additional incentive comes from the knowledge that open source code will be available for people to use and work with "forever." Because code is nearly certain to outlive the developer (or at least to outlive the developer's interest in that specific project), documentation is a means of transferring what the author knows across time as well as space.

But reality is not so generous. In fact open source developers are not always good at documentation, and some of the reasons (in particular, time pressure) are the same as what developers face in a proprietary setting. The culture of open source programming historically has had an intimate relationship to documentation—and like most intimate relationships, it is complicated. In the early days of Unix, programmers learned about the system by playing with it and then talking to Dennis Ritchie or Ken Thompson. Obviously, this approach didn't scale; and as Unix grew in popularity, documentation became increasingly important. Developers' documentation of bugs as well as features and processes eventually became a fundamental principle of Unix and one that was quite novel at the time. Documentation forces programmers to think clearly about what it is they are trying to do. Rewriting code so

it is easier to document, as Ritchie said, is quite characteristic of the Unix culture. Of course, documentation is also a vital part of the scientific, research-oriented tradition in which replicability of methods, as well as results, is considered essential.

7. RELEASE EARLY AND RELEASE OFTEN. User-programmers need to see and work with frequent iterations of software code to leverage fully the potential of the open source process. The evolutionary archetype is not just about voluntary parallel processing among geographically distributed developers; it is also about accelerating the turnover of generations so the rate of error correction can multiply. (Evolutionary biologists know this kind of argument well. Bacteria evolve as quickly as they do for two reasons: their large number *and* the speed with which they reproduce. The first creates a diversity of variation and is the substrate for natural selection; the second is the mechanism for "locking in" to the genetic code that works well and getting rid of what does not.) The open source process in principle mimics this evolutionary strategy, with a feedback and update cycle (at least for larger projects) that is an order of magnitude faster than most commercial software projects. In the early days of Linux, for example, there were often new releases of the kernel on a weekly basis, and sometimes even daily.

But while rapid turnover of generations is tolerable for populations of bacteria (because bacteria can't complain about it), a similar kind of evolutionary process would not be acceptable in software. Rapid evolution is an extremely dynamic process. The vast majority of changes that occur in an evolutionary system are highly *dysfunctional* and they cause the organism to die. That is tolerable for an ecology of bacteria, but not for the ecology of a human-oriented technological system. Of course because changes in software are the result of design, not random variation, a smaller percentage of them are likely to be lethal. But it is still true that one great software feature in an evolutionary "package" of 100 crashing programs would not be an acceptable outcome of a human-oriented development process. More fundamental is that rapid evolution poses the risk of overwhelming the system that selects among variations—and thus introducing errors more quickly than the system can fix them. If this dynamic is set in motion, a system can undergo very rapid deterioration toward the equivalent of extinction through a downward evolutionary spiral.

What open source developers do as individuals, does not guarantee that this will not happen. Put differently, the evolutionary stability of open source software is something that needs to be explained at the macro level because it does not follow directly from the behavior of individual developers.

8. TALK A LOT. Peter Wayner captures something essential about the open source process in this aphorism: How many open source developers does it take to change a light bulb? His answer is, "17. 17 to argue about the license; 17 to argue about the brain-deadedness of the light bulb architecture; 17 to argue about a new model that encompasses all models of illumination and makes it simple to replace candles, campfires, pilot lights, and skylights with the same easy-to-extend mechanism; 17 to speculate about the secretive industrial conspiracy that ensures that light bulbs will burn out frequently; 1 to finally change the light bulb, and 16 who decide that this solution is good enough for the time being."[31]

Open source developers love to talk about what it is they are doing and why. These discussions range from specific technical problems in a project to general issues associated with the politics or business of software development. The email lists for the Linux kernel are enormous and bubble with activity. Beyond Linux, there are huge lists on Slashdot, Kuro5hin, Freshmeat, and other popular websites.[32] People talk about projects in progress, about new ideas, about old bugs, about new hardware, about the politics of antitrust suits against Microsoft; almost nothing seems off limits. Some of these discussions are tightly organized around a specific technical problem and are clearly aimed at gaining consensus or defending an argument about how to proceed with a project. Others are general opinion-venting or discussions about the merits and demerits of the open source development process.

"Talking" among open source developers does not mean calm, polite discussion. One of the common and most misleading fallacies about the open source process is that it involves like-minded geeks who cooperate with each other in an unproblematic way because they agree with each other on technical grounds. Even a cursory glance at the mailing lists shows just how wrong this concept is. Discussion is indeed generally grounded in a common belief that there exist technical solutions to technical problems, and that the community can see good

code for what it is. But this foundation of technical rationality is insufficient to manage some of the most important disagreements. It works fairly well to screen out arguments that are naïve or have little technical support. And it tends to downplay abstract "good" ideas unless and until there is some actual code to talk about. But it does not cleanly define problems, identify solutions, or (most importantly) discriminate up front among contending strategies for solving problems.

And technical rationality hardly restricts the tone of the conversation. When open source developers disagree with each other, they do not hold back. They express differences of opinion vehemently and vent their frustrations with each other openly. Even by the relatively pugnacious standards of contemporary academic discourse, the tone of exchange is direct and the level of conflict, at least in language, is quite high. Torvalds set the standard for this kind of behavior in Linux mail lists when in 1992 he wrote to Andrew Tanenbaum (the author of Minix): "Linux still beats the pants off Minix in almost all areas . . . your job is being a professor and a researcher, that's one hell of a good excuse for some of the brain-damages of Minix."[33]

How Do Open Source Developers Collaborate?

All complex production processes face a problem of collaboration. Individuals make efforts, but they need to work together, or at least their contributions need to work together. The open source process is no different. To get past the boundary where the complexity of software would be limited by the work one individual programmer can do on his own, the development process has to implement its own principles of collaboration. To explain the collaboration principles and mechanisms of the open source process is to explain the guts of that process. But first the problem needs to be described more accurately.

It is common to see open source collaboration explained away with a slogan like "the invisible hand" or "self-organizing system." But these are not very useful descriptions and, for the most part, they obfuscate the explanatory issue more than they illuminate it. What do they really mean? The term "invisible hand" is a placeholder for an argument about coordination by price signals, which is supposed to happen in markets. The term "self-organizing system" is a placeholder for an argument about how local forces, those that act between nearby agents, sum to global or at least greater-scale organization. When used carelessly, both often really mean, "I don't understand the principles of organization that facilitate collaboration."

It is better to drop both these notions for now. I am not assuming they are wrong. I am simply taking the position that any argument about principles of collaboration in open source should be built from the ground up, relying on a careful description of actual behavior rather than assumed from abstract principles. Given that, there are three important aspects of behavior to describe in this chapter: the use of technology, the development of licensing schemes, and the emergent similarities between the configuration of technology, and the social structures that create it.

TECHNOLOGY IS AN ENABLER. Networking has long been an essential part of the open source development process. Before computer-to-computer communications became common, prototypical open source communities grew up in spaces bounded by geography. The main centers in the United States were Bell Labs, the MIT AI Lab, and UC Berkeley. The density of networks really did fall off with something approximating the square of the distance from these geographic points. Extensive sharing across physical distances was difficult, expensive, and slow. It was possible for programmers to carry tapes and hard drives full of code on buses and airplanes, which is exactly what happened, but it was not very efficient.

The Internet was the key facilitating innovation. It wiped away networking incompatibilities and the significance of geography, at least for sharing code. As a result, the Internet made it possible to scale the numbers of participants in a project. There are downsides to working together virtually: The transferring of tacit knowledge at a water cooler is a reminder that face-to-face communication carries information that no broadband Internet connection can.[34] But the upside of TCP/IP as a standard protocol for communication was huge because it could scale the utility of electronic bandwidth in a way that physical space could not. Put 25 people in a room and communication slows down, whereas an email list can communicate with 25 people just as quickly and cheaply as it communicates with 10 or 250. As the numbers scale and the network grows, the likelihood of proliferating weak ties—that is, pulling into the process people with very different sets of expertise and knowledge—goes up as well.

To simply share code over the Internet became a seamless process.

As bandwidth increased over time, the Internet also enabled easy access to shared technical tools, such as bug databases and code-versioning systems, that further reduced the barriers to entry for user-programmers.

The more complicated issues, such as communication about the goals of a project or working out disagreements over directions to take, are not seamless over the Internet. In principle the Internet ought to reduce the costs (in a broad sense) of coordinating by discussion and argumentation rather than by price or corporate authority. In practice there is really no way to measure the overall impact because the costs are paid in such different currencies. What practice does reveal is that open source developers make enormous use of Internet-enabled communications to coordinate their behavior.

LICENSING SCHEMES AS SOCIAL STRUCTURE. Another pernicious myth about open source developers is that they are hostile to the concept of intellectual property rights. Actually, open source developers are some of the most vehement defenders of intellectual property rights. Rarely do these developers simply put their software in the public domain, which means renouncing copyright and allowing anyone to do anything with their work.[35] Open source collaboration depends on an explicit intellectual property regime, codified in a series of licenses. It is, however, a regime built around a set of assumptions and goals that are different from those of mainstream intellectual property rights thinking. The principal goal of the open source intellectual property regime is to maximize the ongoing use, growth, development, and distribution of free software. To achieve that goal, this regime shifts the fundamental optic of intellectual property rights away from protecting the prerogatives of an author toward protecting the prerogatives of generations of users.

The basic assumptions behind open source is that people want to be creative and original and they don't need much additional incentive to engage in this manner. The only times when innovation will be "undersupplied" is when creative people are prevented from accessing the raw materials and tools that they need for work. Distribution of raw materials and tools, then, is the fundamental problem that an intellectual property rights regime needs to solve. Solving that problem allows the system to release fully the creative energies of individuals. Even

better, it promises to ratchet up the process over time as a "commons" of raw materials grows. Open source intellectual property aims at creating a social structure that expands, not restricts, the commons.

The regime takes shape in a set of "licenses" written for the most part in the language of standard legal documents. For now, think of these licenses as making up a social structure for the open source process. In the absence of a corporate organization to act as an ordering device, licensing schemes are, in fact, the major formal social structure surrounding open source.

Open source licensing schemes generally try to create a social structure that:

- Empowers users by ensuring access to source code.
- Passes a large proportion of the rights regarding use of the code to the user rather than reserving them for the author. In fact, the major right the author as copyright holder keeps is enforcement of the license terms. The user gains the rights to copy and redistribute, use, modify for personal use, and redistribute modified versions of the software.
- Constrains users from putting restrictions on other users (present and future) in ways that would defeat the original goals.

Precisely how these points are put into practice differs among open source licenses. The differences are core explanatory elements of the open source process. They depend in large part on underlying assumptions about individuals' motivations, and the robustness of the commons, as well as some fundamental quarrels about the moral versus pragmatic values connected to software. BSD-style licenses are much less constraining than is the GPL. Arguments over the "appropriate" way to conceive of and implement licenses are an important part of the story of open source in the 1990s (see Chap. 4). In a very real sense, the open source community figures out its self-definition by arguing about licenses and the associated notions of property, what is worth protecting, that they embody. Remember that licenses act as the practical manifestation of a social structure that underlies the open source process.

The Debian Project, which Ian Murdock started in 1993 to produce an entirely free operating system around a GNU/Linux distribution, is most explicit but characteristic on this point. In 1997 Bruce Perens

(who followed Murdock as the leader of Debian) wrote a document he called the Debian social contract to articulate the underlying ideals.[36] The Debian social contract clearly prioritizes the rights of users, to the point at which it recognizes that many Debian users will choose to work with commercial software in addition to free software. Debian promises not to object to or to place legal or other roadblocks in the way of this practice. The basic principle is nondiscrimination against any person, group of people, or field of endeavor, including commercial use. (There are sharp ethical differences here with at least some free software advocates. These differences became a major point of contention in the late 1990s when Perens and others recast the Debian Free Software Guidelines as "The Open Source Definition," in sharp contrast to the Free Software Foundation's stance against commercial software on principle.) The principle of collaboration at work here is clear: Do nothing to complicate or slow down the widespread distribution and use of open source software. On the contrary, do everything you can to accelerate it by making open source software maximally attractive to users. This is intellectual property to be sure, but it is a concept of property configured around the right and responsibility to distribute, not to exclude.

ARCHITECTURE TRACKS ORGANIZATION. More than thirty years ago, Melvin Conway wrote that the relationship between architecture and organization in software development is driven by the communication needs of the people who are trying to collaborate.[37] Conway's Law argues that the structure of a system—in this case, a technological system of software code—mimics the structure of the organization that designed it. Because the point of organization ultimately is to facilitate successful coordination of the technology development process, Conway's Law has been interpreted to mean that the technology architecture should drive thinking about the organization, not vice versa.[38]

The problem is that early formulations of software architecture are best guesses and are likely to be unstable, while a formal organization set up to support those guesses locks in quickly and is hard to change. As the architecture evolves, new communication paths are necessary for collaborative work to succeed, but those communication paths are not hardwired into the organization. This is one reason (not the only one, of course) why informal, unplanned communication is so critical

within organizations. It is not just tacit knowledge that gets passed around at the water cooler; it is also communicable knowledge that would travel through standard pathways quite easily, if those pathways did exist.

Herbsleb and Grinter documented some of the ways in which these informal knowledge transfers become more difficult with distance and physical isolation, regardless of Internet connections.[39] The existence of a formally structured organization can, ironically, exacerbate the problem. The formal organization puts a stake in the ground and marks out particular communication paths, which makes it more awkward to step outside those paths. Developers sometimes say they feel like they are working in silos. When they need to talk to someone in another silo, the initial difficulty comes in knowing exactly whom to contact. The next is the difficulty of initiating contact and then following up. This is a familiar feeling for anyone who has wondered how to interpret an unreturned phone call or email when the other party is a stranger (did the person not receive my message? Is she on vacation? Does she just not care?)

In commercial software development, this silo problem causes more than just social awkwardness. Developers communicate outside their silos less frequently than they should; they are inclined to take a risk that problems will not arise. Furthermore, developers in other silos say they are not consulted frequently enough on decisions that affect what they do. When communication does traverse silos, it takes longer to find the right contact and then even longer to solve problems (what developers call "cycle time"). Disagreements that cross silos frequently have to be escalated to higher management for resolution.[40] The really interesting observation is the way these communication problems reflect themselves back into the code—how the organization comes to influence the architecture. At least some of Herbsleb and Grinter's developers reported that they "strove to make absolutely minimal changes, regardless of what the best way to make the change would be, because they were so worried about how hard it would be to repair the problem if they 'broke the system.'"[41] Whatever their technical predilections, developers are clearly going to be influenced to write code that compensates for the imperfections of the organizational structure that sets the parameters for collaboration.

Open source developers know this problem well. Because their or-

ganization is voluntary and most often informal, Conway's Law makes extraordinary demands on the technological architecture. This is one of the major reasons why technical rationality is not deterministic in the open source process. Technical rationality always is embedded in a cultural frame, which for open source generally means Unix culture. Technical rationality also is embedded in the organizational characteristics of the development model. When people talk about "clean" code and so on, they are making statements not only about some distinct characteristic of source code but also about the way in which the technical artifact can interface with and be managed by a particular organized community.

Open source developers often say, "Let the code decide." This sounds on the face of it like an unproblematic technical rationality, but it is not so in practice. The most important technical decisions about the direction of software development are those that have long-term consequences for the process of development. Many imply a set of procedures that will need to be carried out in the development path going forward. Implicitly then, and often explicitly, technical decisions are influenced by beliefs about effective ways to organize development. Technical discussions on how things should work and should be done are intimately related to beliefs about and reflections on social practices. Modularization of source code is an intimate reflection of the complex collaboration problem invoked by voluntary large-scale parallel processing in open source development. Technical rationality may be a necessary part of the foundation for the open source process, but it is not sufficient.

How Do Open Source Developers Resolve Disagreements?

Anyone who has dabbled in software engineering recognizes that disagreement is the rule. A large number of very smart, highly motivated, strongly opinionated, often brazenly self-confident, and deeply creative minds trying to work together creates an explosive mix.

Successful collaboration among these highly talented people is not simple. Conflict is customary. It will not do to tell a story about the avoidance of conflict among like-minded friends who are bound together by an unproblematic technical rationality, or by altruism, or exchanges of gifts.[42] The same bandwidth that enables collaboration on the Internet just as readily enables conflict. People could use the Internet to break off and create their own projects, to skewer the efforts of others, and to distribute bad code just as quickly and widely as good code. They do use the Internet on a regular basis to argue with each other, sometimes quite bitterly. What needs to be explained is not the absence of conflict but the management of conflict—conflict that is sometimes deeply personal and emotional as well as intellectual and organizational.

Major conflicts within the open source process seem to center on three kinds of issues.[43] The first is who makes the final decision if there are deep and seemingly irreconcilable disagreements over the goodness of a specific solution, or a general direction for dealing with a problem. The second is who receives credit for specific contributions to a project. (Ironically, this second source of conflict can become worse in more successful collaborations, because much of what is good in these collaborations is created in the context of relationships as much as by any particular individual.) The third major source of conflict is the possibility of forking. The right to fork *per se* is not at issue. What causes contention is the issue of legitimacy. It is a question of who can credibly and defensibly choose to fork the code, and under what conditions.

Similar issues arise when software development is organized in a corporate or commercial setting. Standard theories of the firm explain various ways in which these kinds of conflicts are settled, or at least managed, by formal authoritative organizations. Most of these mechanisms are just not available to the open source community. There is no boss who can implement a decision about how to proceed with a project; there is no manager with the power to hire and fire; and there is no formal process for appealing a decision.

In open source much of the important conflict management takes place through behavioral patterns and norms. There are two descriptive elements of these norms that I consider here: the visible nature of leadership and the structures of decision-making.

Leadership is a peculiar issue for the open source community. The popular media as well as most extended treatments of open source focus on one project—Linux—and its remarkable leader, Linus Torvalds. There certainly is something unique about the man as an individual. His style of leadership is alternatively charismatic and self-dep-

recating. Torvalds (surprisingly to some) is a shy person whose self-effacing manner seems authentic, not manufactured for effect. Developers respect Torvalds for having started Linux, but much more so for his extraordinary intellectual and emotional commitment to the Linux project through graduate school and later through a Silicon Valley programming job. Although he does not claim to be the very best programmer, he has maintained a clear vision about the evolving nature of Linux, as well as the structure and style of the code that he incorporates into the kernel; and that vision has turned out over time to look "right" more often than not. His vision has never been enforced in an aggressively authoritative way. When challenged about his power over Linux early on, Torvalds posted to the Linux mail list (on February 6, 1992) this revealing comment: "Here's my standing on 'keeping control,' in 2 words (three?): I won't. The only control I've effectively been keeping on Linux is that I know it better than anybody else."

In fact, one of the most noteworthy characteristics of Torvalds's leadership style is how he goes to great lengths to document, explain, and justify his decisions about controversial matters, as well as to admit when he believes he has made a mistake or has changed his mind. Torvalds seems intuitively to understand that, given his presumptive claim on leadership as founder of the Linux project, he could fail his followers in only one way—by being unresponsive to them. That does not in any way rule out disagreement. In fact it prescribes it, albeit within a controlled context. In the end, Torvalds is a benevolent dictator, but a peculiar kind of dictator—one whose power is accepted voluntarily and on a continuing basis by the developers he leads. Most of the people who recognize his authority have never met him and probably never will.

But Torvalds's charismatic leadership style is clearly not the only way to lead an open source project. Richard Stallman has a very different leadership style that has developed from his extraordinary prowess as a code writer. He is self-consciously ideological and (in sharp contrast to Torvalds's fervent pragmatism) sees his leadership role at the Free Software Foundation as piously defending an argument about ethics and morality. Brian Behlendorf, one of the central figures behind the Apache web server, has yet another leadership style and is known for engaging deeply in the development of business models around open source software. This kind of variance does not demonstrate that leadership is irrelevant; instead, it suggests that there are different ways to lead and that a satisfying explanation of the open source process needs to go beyond the question of leadership.

There is just as much variance in decision-making structures for open source projects. In the early days of Linux, Linus Torvalds made all the key decisions about what did or did not get incorporated into the kernel. Many small-scale open source projects are run this way, with one or a few decision makers choosing on the basis of their own evaluations of code.

Chapter 4 describes how the decision-making system for Linux was restructured in the mid-1990s, as both the program itself and the community of developers who contributed to it grew enormously. Linux today is organized into a rather elaborate decision-making structure. Torvalds depends heavily on a group of lieutenants who constitute what many programmers call "the inner circle." These are core developers who essentially own delegated responsibility for subsystems and components. Some of the lieutenants onward-delegate to area-owners (sometimes called "maintainers") who have smaller regions of responsibility. The organic result looks and functions like a decision hierarchy, in which responsibility and communication paths are structured in something very much like a pyramid. Torvalds sits atop the pyramid as a benevolent dictator with final responsibility for managing disagreements that cannot be resolved at lower levels. The decision hierarchy for Linux is still informal in an important sense. While programmers generally recognize the importance of the inner circle, no document or organization chart specifies who is actually in it at any given time. Everyone knew for years that the British programmer Alan Cox was responsible for maintaining the stable version of the Linux kernel (while Torvalds spends more of his time working on the next experimental version) and that Torvalds pro forma accepted what Cox decided. This made Cox close to something like a vice-president for Linux. But Torvalds did not handpick or formally appoint Cox to this role; he simply took it on as he established his expert status among the community over time.[44]

The BSD derivatives on the whole follow a different decision-making template, organized around concentric circles. A small core group controls final access to the code base. This group grants (or revokes) the rights to the next concentric circle, who can modify code or com-

mit new code to the core base. These are the "committers." In the third concentric circle are the developers, who submit code to committers for evaluation. The boundaries of the circles are generally more definite: FreeBSD, for example, has a core of 16 and about 180 committers in the second circle.

Larry Wall, the originator of the programming language Perl, in the mid-1990s developed a different version of a delegated decision-making structure.[45] There is an inner circle of Perl developers, most of whom took an informal leadership role for a piece of the code during the transition from Perl version 4 to Perl version 5. Wall would pass to another developer the leadership baton, that person would work on a particular problem and then release a new version of the code, and then pass the baton back to Wall. This process developed into what Wall called "the pumpkin-holder system."[46] The pumpkin holder acts as chief integrator, controlling how code is added to the main Perl source. In a kind of rotating dictatorship pattern, the pumpkin gets passed from one developer to another within the inner circle.

Apache has evolved a more highly formal decision-making system. The Apache Group is an elite set of developers that together make decisions about the Apache code base. The Group began in 1995 with eight core developers who worked together to build Apache out of a public domain http daemon, which is a piece of server software that returns a web page in response to a query. There was no single project leader *per se*, and the group was geographically diverse from the start, with core developers in the United States, Britain, Canada, Germany, and Italy. The Apache Group devised a system of email voting based on minimal quorum consensus rule. Any developer who contributes to Apache can vote on any issue by sending email to the mailing list. Only votes cast by members of the Apache Group are binding; others are simply expressing an opinion. Within the Apache Group, code changes require three positive votes and no negative votes. Other decisions need a minimum of three positive votes and an overall positive majority. The Apache Group itself expands over time to include developers who have made excellent and sustained contributions to the project. To join the Apache Group, you must be nominated by a member and approved unanimously by all other members.[47]

Each of these decision-making systems has strengths and weaknesses as coordination mechanisms. As Conway's Law suggests, they make dif-

ferent demands on the technology architecture. What they share is the fundamental characteristic of the open source process—there is no authority to enforce the roles and there is nothing to stop an individual programmer or a group of programmers from stepping outside the system. On a whim, because of a fundamental technical disagreement, or because of a personality conflict, anyone could take the Linux code base or the Apache code base and create their own project around it, with different decision rules and structures. Open source code and the license schemes positively empower this option. To explain the open source process is, in large part, to explain why that does not happen very often and why it does when it does, as well as what that means for cooperation.

This chapter painted a picture of the open source process, the problem(s) it is trying to solve, and what we can recognize about how it seems to do that. These functional characterizations together describe a set of important interactions among the developers who create open source software—what they do, how they work together, and how they resolve disagreements. Clearly these do not constitute by themselves a robust explanation, and I have pointed out at various junctures why not. To answer the broader question with which I began this chapter—what are the conditions or boundaries for the open source process in software engineering, or for extending a version of that process to different kinds of production?—we need a more general and deeper explanation. That explanation needs to elucidate more precisely the basis of the equilibrium "solution" that open source has found, and to illustrate either why it is not challenged or why challenges do not disrupt it. The next chapter returns to a historical account of open source software in the 1990s. Chapters 5 and 6 build the explanation.

CHAPTER 8

The Code That Changed the World?

Like many elements of the Internet economy, the media surrounds open source software with an overblown mix of hype and cynicism. These are short-term distractions from a profound innovation in production processes. Consider an analogy from *The Machine that Changed the World*, a study of the Toyota lean production system for manufacturing autos. That book made two simple and profound points: The Toyota "system" was not a car, and it was not uniquely Japanese. The parallels are obvious. Open source is not a piece of software, and it is not unique to a group of hackers.

Open source is a way of organizing production, of making things jointly. The baseline problem is that it is not easy for human beings to work together and certainly not to produce complex integrated systems. One solution is the familiar economy that depends on a blend of exclusive property rights, divisions of labor, reduction of transaction costs, and the management of principal-agent problems. The success of open source demonstrates the importance of a fundamentally different solution, built on top of an unconventional understanding of property rights configured around distribution. Open source uses that concept to tap into a broad range of human motivations and emotions, beyond the straightforward calculations of salary for labor. And it relies on a set of organizational structures to coordinate behavior around the problem of managing distributed innovation, which is different from division of labor. None of these characteristics is entirely new, unique to open source, or confined to the Internet. But together,

they are generic ingredients of a way of making things that has potentially broad consequences for economics and politics.

This book explains the open source process—how it came to be, how it operates, how it is organized. I put forward a compound argument about individual motivations, economic logic, and social structure to account for a process that reframes the character of the collective action problem at play. This argument yields insights into the nature of collaboration in the digital political economy. It also yields insights about how human motivations operate in that setting and the possibilities for conflict management that result. At a minimum, these are pragmatic demonstrations about a production process within, and as a product of, the digital economy that is quite distinct from modes of production characteristic of the predigital era. There are broader possibilities. The key concepts of the argument—user-driven innovation that takes place in a parallel distributed setting, distinct forms and mechanisms of cooperative behavior regulated by norms and governance structures, and the economic logic of "antirival" goods that recasts the "problem" of free riding—are generic enough to suggest that software is not the only place where the open source process could flourish.

There are many things that this book does not try to be. It is not a detailed ethnography of the communities that build open source software. It is not an extended analysis of the practical business and management issues that open source raises. It is not a book about theories of innovation, intellectual property rights, or copyright *per se* (although I touched on each of these). And it is not a claim that open source is necessarily a good thing, a morally desirable thing, or an antidote to undesirable things that are happening in the information economy.[1] The open source story can be mined for insights and arguments as well as normative claims in each of these areas as well as others, and I hope that these disclaimers will be seen as an invitation to others.

For a political economy audience in particular, I need to emphasize that this book also is not an attack on rational choice models of human behavior and organization. There are strong intellectual, disciplinary, and even emotional attachments to rational choice explanations, about which I am frankly agnostic. In my view it is the substantive content of the terms in the explanation that matters, not how it is or can

be labeled. I do not see evidence for an explanation based on altruism, which (in some formulations, but of course not all) stands opposed to rational behavior. A minimal and reasonable standard of rational behavior to me means that individuals respond to their environment, compare the costs and benefits of a behavioral choice, and will not choose to do actions that have more costs than benefits.[2] That is simply a way of asking, What are their motivations, how do they make sense of what they are doing, and how does it add up to a choice, individual and social? Clearly, a great deal of human behavior is not motivated by narrow rationality concerns. The vast majority of human behavior is never monetized. Most art is not sold but simply given away. Only a tiny proportion of poetry is ever copyrighted or published. And most software code has never and will never see the inside of a shrink-wrapped box.

At the same time, computing and the creation of software have become deeply embedded in an economic setting. The money stakes are huge. The Free Software Foundation (among others) condemns this fact from a moral perspective, but that does not make it untrue. An uneconomic explanation of open source, if such a thing really were possible, would probably be uninteresting and also probably wrong. The challenge is to add substantive meaning to a rational and economic understanding of why the relevant individuals do what they do, and how they collaborate, in open source.

The perspective I have adopted is limited in these ways, but it does yield some interesting implications. I begin this concluding chapter with a set of ideas about the political economy of the Internet, concentrating particularly on the logic of distributed innovation, and the meaning and function of the commons in economic and social life. I then consider some general lessons about cooperation, how power relates to community structures, and the next phase of information technology–enabled economic growth in both developed and developing economies. The final sections take up two important issues that open source raises for political economy and particularly international politics. The first is about conceptualizing what happens at the interface between the hierarchy and the network. And the second is about hypothesizing the scope of the implications. How generalizable is the open source process and how broad might its repercussions turn out to be?

My answer, to foreshadow, has a mini-max flavor to it. Even if this were simply a story about software, the consequences are significant for the next phase of global economic growth. For international politics, the demonstration of large-scale nonhierarchical cooperation is always important; when it emerges in a leading economic sector that disregards national boundaries, the implications are potentially large. And that is the minimum case. If the open source process is a more generalizable production process that can and will spread, under conditions that I lay out as hypotheses, then the implications will be bigger still.

Rethinking Property

Open source is an experiment in social organization around a distinctive notion of property rights. Property is, of course, a complicated concept in any setting. For the moment, consider a core notion of property in a modern market economy as a benchmark. The most general definition of a property right is an enforceable claim that one individual has, to take actions in relation to other individuals regarding some "thing."[3] Private property rests on the ability of the property holder to "alienate" (that is, trade, sell, lease, or otherwise transfer) the right to manage the "thing" or determine who can access it under what conditions. Put most simply, ownership of property is the right to exclude others from it according to terms that the owner specifies.[4] This is the core entitlement that energizes the act of market exchange, with all of its attendant practices. Most of the time we are inclined to think of property in this sense and the exchange economies through which it flows, as almost a natural form of human interaction. The most obvious applications of this story are to physical widgets, but it is in no sense limited to heavy objects you can drop on your foot. With special (and limited) exceptions for things like fair use of copyrighted material, this same core concept applies to mainstream ideas about intellectual property.

Lawyers know that all of these ideas about property are problematic, but those controversies should not be taken as evidence that the underlying notions are less important in practice and in theorizing about social systems. To some big-picture economic historians like Douglass North, the creation and securing of core property rights enable the

development of modern economic growth paths; their presence or absence explains why some societies are wealthy and others are poor.[5] In international relations theory, one of the essential functions of the sovereign state is to secure exclusive property rights. John Ruggie has argued that the extrusion of a universal realm along with overlapping authorities and rights lies at the heart of the historic transformation from the medieval to the modern era.[6] And in contemporary controversies around intellectual property, the notion of exclusion remains like a brick wall that other arguments keep bumping their heads up against. When you clear away all the complexity, isn't it somehow "natural" that property carries with it the right to exclude others from it, or to sell it or trade it to others in a way that transfers that right to them?[7]

Open source radically inverts this core notion of property. Property in open source is configured fundamentally around the right to distribute, not the right to exclude. If that sentence still sounds awkward, it is a testimony to just how deeply embedded within our intuition the exclusion view of property really is. With all its shortcomings, humans have built a workable economy on a foundation of property rights as exclusion. The open source process poses a provocative question: Is it possible to build a working economic system around property rights configured as distribution, and what would such a system look like? Much of this chapter tries to sketch out pieces of an answer to that question.

Recognize that open source–style conceptions of property are not at all unique to the software world. I am not talking here about the notion of gift economies.[8] I am talking about the many parts of human life that are organized around property notions more closely connected to *stewardship* or *guardianship* than to exclusion, and in particular to stewardship combined with distribution. As I said earlier, there is no state of nature in how people own things or what that means. Property is a socially constructed concept that can be and is constructed in different ways. As an example, consider modern religious traditions in the contemporary United States. What does a rabbi own, or more precisely, what is his or her property relationship to a tradition? Religious leaders at one time held essentially exclusionary keys to traditions. The Jerusalem Temple locked up the Torah within a holy of holies; only the highest priests could enter. Blessings, sacraments, and rituals were under Rabbinic control. Excommunication as it was practiced in

the Catholic Church is the essence of an exclusionary property right. I am less concerned here with how these rights changed than with the stark differences they exhibit today.[9] Certainly a rabbi no longer owns a tradition in the sense that he or she can exclude you from taking a piece of it, modifying it, and combining it with other traditions in ways that suit your own needs. I have a friend who lights Shabbat candles but on Saturday night, practices Buddhist meditation, and believes very deeply in some of the core teachings of Jesus Christ. That is not exceptional; rather it illustrates the way in which people now treat notions of property around religious traditions. In effect, the property rights regime is much like open source. People take the religious "code," modify it, recombine it with pieces of code from elsewhere, and use the resulting product to scratch their spiritual itch.

It is only one small step from there to redistributing what you have created to others as a "new" religion, which is increasingly a common practice. Many established religious leaders resist these changes—understandably, because the changes profoundly threaten their basis of legitimate authority. But it is probably too late to go backward. There is too much "open code" in circulation. Technology surely enabled this change (it does matter that anyone can access any religious text in an instant on the web). But more profoundly important is the mindset change around what property rights attach to texts and traditions, which are often intricately intertwined. It is this property mindset shift that drives change in what a rabbi is. More generally, there has been a dramatic change in what it means to be a leader in a religious community. Change the foundations of property, and you change the network of relationships that radiate outward from that which is owned, in fundamental and often unexpected ways.

I will come back to this point later when I discuss power and communities; but for the moment, recognize an interesting corollary. In nonauthoritative settings—that is, in relationships primarily characterized as bargaining relationships—power derives in large part from asymmetrical interdependence.[10] Put simply, the less dependent party to the relationship (the one who would be harmed least by a rupture) is the more powerful. In the kinds of modern religious communities I am talking about here, it is the leader who is dependent on the followers more than the other way around. This dependence changes the dynamics of power. Leaders of open source projects understand the con-

sequences intuitively—the primary route to failure for them is to be unresponsive to their followers. In Hirschman's terms, this is a community that empowers and positively legitimates exit while facilitating voice.[11] Understanding how leadership will function in a community like that is a work in progress, for the open source community, religious communities, and—just possibly—other elements of human organization as well.

How far, then, might this technologically enabled, mindset-shifting change in property rights extend? Clearly similar issues are already in play within controversies over copyright and how it is being remade in the digital era.[12] What is at stake here is more fundamental than moving the fuzzy boundaries around the idea-expression distinction or the precise parameters for what constitutes fair use. It is, instead, the logical basis of copyright as an institution. For property fundamentalists, who believe that the right to exclude others ultimately is a moral consequence of having created something, transaction cost–reducing technological innovations may shift the boundaries of fair use (often toward greater constraint), but the core foundations of intellectual property are not at risk.[13] Property fundamentalists are a minority; most theorists see copyright as a social bargain anchored in a pragmatic compromise. Particularly with digital technologies, it is inefficient to limit distribution (because the costs of nonrival distribution essentially are zero) once a digital good has been created. The pragmatic compromise was, How much excludability is needed to generate rents that can incentivize people to create in the first place?

The weights on the terms of that compromise have long been controversial, but really only on the margins (is 25 years long enough? Why not 14 or 154?). The fact is, no one really knows the terms of a creative person's utility function or how to measure the social utility that is gained from easing distribution, so the answer is a form of guesswork—or in the political process, a game of power and vested interests. Now new communities have put the basic terms of the compromise itself up for grabs, and the questions are an order of magnitude more significant. Napster was the most attention-grabbing example. What made Napster important was not its underlying technology, but rather how Napster used technology to experiment with a fundamentally different notion of intellectual property. Rather than incentivizing creativity by limiting distribution, Napster incentivized distribution and assumed that creativity would take care of itself. The record com-

panies feared Napster not because of the technology (it's too late to do anything about that), but because of the mindshift that the experiment represented. The courts cut short the Napster experiment, but there will be others. In a real sense, open source software is a broader and longer-standing experiment in the same kind of logic. And it has worked extremely well, as the early evidence suggested Napster was doing.

These experiences imply something about property that is both simple and profound. The problem is not just that the current regime is conceptually insecure. The problem is that technology has made visible the fact that it does not work very well in practice, and that other things that are not supposed to work, do. If we find out from these kinds of experiments that some core assumptions about formulating the balance between creativity and distribution were wrong—that is, not just the weight of the terms in the bargain, but the structure of the bargain itself—then the debates over copyright extension to 50 or 100 years, or whether fair use means 1 or 20 photocopies, will seem quaint. Property rights could shift dramatically because (fundamentalism and vested interests in the political process aside), from an intellectual standpoint, there is nothing to anchor them where they are except utilitarianism. I will come back to some of the practical consequences of that for cooperation and institutions.

Organizing for Distributed Innovation

Adam Smith was a profound theorist of productivity gains from the division of labor. What then limits the degree of granularity in that division and consequent specialization? For Smith, it was mainly the extent of the market. Smith observed more finely developed divisions of labor in cities than in rural settings, and he reasoned that this was because markets were larger in cities. Since Smith, declining transportation costs have raised the effective size of markets in world trade, thereby making other limiting factors more visible.[14] Gary Becker and Kevin Murphy explicitly summarized some of the costs in a modern economy of combining the efforts of specialized workers: principal-agent problems, the possibilities for free riding, and, most deeply, the problem of coordination among complex pieces of knowledge that no one person can hold in totality.[15]

As I explained in Chapters 5 and 6, the open source process re-

frames the problem of free riding in fundamental ways. And distributed innovation on the open source model does not depend so heavily on bringing agents' incentives into line with those of principals, mainly because the differentiating lines between agents and principals are made so blurry. The problem of coordinating complex pieces of knowledge then becomes the key rate-limiting factor in productivity. This is not news: Friedrich Hayek understood it to be the essential problem of a "modern" economic order. He believed deeply that the price mechanism was the way to make it work.[16] Since Hayek, knowledge management theorists have raised serious doubts about the robustness of the price mechanism for coordinating complex knowledge systems. The Internet, because of the way in which it organizes intelligence and control, both makes this question of how to coordinate complex distributed knowledge more immediate, and presents partial solutions to it.

The essence of the Internet lies in an engineering principle that reflects a simple design decision by its original architects. David Isenberg calls it "the stupid network"; other terms are "neutral" network or end-to-end network.[17] Because the Internet was configured originally as a network of multiple, diverse networks, and because the architects of the Internet decided not to try to anticipate how their network might be used, they designed the protocols that are today's Internet to be neutral or "stupid." The Internet does not know what it is carrying. The intelligence and the majority of the processing power in the network are pushed out to the edges.[18] This is opposite to the switched telephone network that is made up of relatively stupid edges (your phone) and a smart network (complicated switches at the center) that make the system work by having intelligence at the center.

In networks, intelligence is a source of power and control. The phone companies control the switched telephone networks because they own the switches where the intelligence lies. In the Internet, power follows intelligence out to the edges. Barriers to entry are low and nondiscriminatory. The conduit becomes just a pipe that carries what people on the edges ask it to carry. The stupid network empowers innovators to try out their innovations without having to ask permission or procure a license from the network owner. And it is a network in which feedback moves directly from user to innovator without being filtered through a network-owning intermediary.

The open source process and the Internet share central features of an end-to-end conceptual architecture. Distributing source code and licensing it under GPL ensures low and nondiscriminatory barriers to entry. The decision about when and how to innovate then lies with the user on the edge of the network. The center does not really control the process so much as incorporate pieces of innovation into itself. And if it fails to do so successfully, a new center can always form at what was formally an edge.

Modern communications networks have done what Smith said. By reducing transaction costs and expanding the size of potential markets, they have driven more elaborate and complex divisions of labor in production. With massively reduced transaction costs, you can easily imagine breaking up a product into smaller and smaller modules and assigning each to the most efficient producer anywhere in the world.[19] Speed and efficiency go up, and costs go down. This division of labor could become very elaborate, but it would still be a division of labor in a fundamental sense. A central decision-making structure runs the process, divides and assigns tasks, and puts back together the results. Its major problem is the same as in any division of labor—getting the least efficient link in the chain to deliver.[20]

End-to-end innovation goes a step beyond simply reduced transaction costs. It enables parallel processing of a complex task in a way that is not only geographically dispersed but also functionally dispersed.[21] End-to-end architecture takes away the central decision maker in the sense that no one is telling anyone what to do or what not to do. This is the essence of *distributed innovation,* not just a division of labor. There are no weak links in this chain because there is, in a real sense, no chain. Innovation is incentivized and emerges at the edges; it enters the network independently; and it gets incorporated into more complex systems when and if it improves the performance of the whole.

That is of course an idealized version of the story. An end-to-end technological architecture may indeed be truly neutral, but technology does not exist on its own. The political and organizational interfaces that touch that technology and link it to human action typically are biased. Pricing of the Internet is a political construction, a complex legacy of telecommunications regulation that allowed a form of arbitrage on top of existing infrastructure to favor packet-switched communications. In contrast, legal structures that support large-scale

production in corporate hierarchies may be biased against distributed innovation.[22] As open source software becomes increasingly mainstream in corporate applications, people have started companies to customize and service the software, and these companies have to make money, follow corporate law, and otherwise interface with conventional economic and legal systems. These constraints require (and generate) some real organizational innovation, which becomes a necessary piece of sustaining end-to-end innovation systems.

This point may seem obvious, but it is something that technology mavens often sidestep. Eric Raymond famously said about open source, "with enough eyeballs all bugs are shallow." What is clearly missing from that statement, and is ultimately as important, is how those eyeballs are organized. What happens in end-to-end systems (and the characteristics of distributed innovation) depend heavily on organization. An end-to-end network architecture does not lead directly to the social equivalent, an end-to-end organization of the institutions that use networks as tools.

The underlying problem here is to specify general organizational principles for distributed innovation, a problem-solving algorithm that works. The open source process suggests that there are four principles:

- *Empower people to experiment.* This is a familiar concept. To make it work depends on technology (people need easy access to tools) and on socially constructed incentives.
- *Enable bits of information to find each other.* This is an engineering concept. A diverse set of experiments will produce a little signal and lots of noise, and in many different places. The bits of information that are signal need to be able to find each other and to recognize each other as signal.
- *Structure information so it can recombine with other pieces of information.* This is also an engineering concept, an extension of the notion of modularization. Signals need essentially to be in the same "language" so, when they find each other, they can recombine without much loss of information.
- *Create a governance system that sustains this process.* This is a social concept, often (but not necessarily) expressed in rules and law. The GPL prevents private appropriation of a solution, which is an important part of what governance needs to do. Governance of a distributed innovation system also needs to scale successfully (organized effectively, more eyeballs really are better than fewer). And it needs to do this at relatively low overhead costs.

Looking at this list of principles, you have to be struck by its resemblance to the algorithm represented by the genetic code. DNA "experiments" through easy mutation; pieces of information find each other through chemical bonding; the process of reproduction is exquisitely engineered to enable recombination. And there are elaborate internal governance systems—repair enzymes, immune responses, regulatory genes—that allow the system to scale surprisingly well.[23] I don't want to push this analogy too far. But because I have put more stress on the advantages, I do want to use it to suggest some of the downsides that also come with distributed innovation.

Consider, for example, the well-known problem of how systems subject to path dependency through increasing returns can lock in to suboptimal equilibrium.[24] One way to read the story of Linux is to say that the open source community has found a way to make the jump from the equivalent of QWERTY to DVORAK.[25] On that logic, other operating systems are suboptimal performers but are locked in to markets by increasing returns and high costs of switching. An aggregate jump to a more optimal performer would bring better returns in the long run but is hard to engineer due to the massive upfront investment required that no one has the incentive to provide (and the incumbent has great incentives to block). Coordinating the investment jump among all the people who would have to make it to make it worthwhile is impossibly expensive, even if there were a clear focal point for the alternative equilibrium. The system remains stuck in a suboptimal path.

But Linux is challenging that equilibrium and may overturn it in at least some market segments. If this happens, open source will have demonstrated one way to invest successfully in the jump to a higher performance path. Traditional economic analysis will suggest that this was the outcome of subsidization—after all, Linux is essentially free up front and thus the community of developers heavily subsidizes the jump. That analysis is not wrong, so far as it goes. But we should be careful not to accept typical related assumptions about the reason that subsidy is being offered. These developers are not making standard

calculations about longer-term returns nor are they providing subsidies at time t=0 in the interest of later exploiting monopoly power at time t=1. Indeed, the form of the subsidy (giving away the source code) is nearly a binding commitment *not* to do that.

This sounds like good news, but there is a potential downside to this logic as well. Could an operating system market dominated by Linux get stuck over time in its own suboptimal equilibrium? The open source process is very good at debugging, but debugging an increasingly inefficient architecture at some point can become an exercise in futility. Old buildings can be renovated in many different ways, but at some point it really does make economic and aesthetic sense to knock them down and start over.[26] The winner-takes-all positive network externalities of software make this a hard problem, whether we are in a proprietary or open source setting.[27] In principle, it may be harder, not easier, for a nonhierarchical open source community to move to a fundamentally new architecture, and there may come a time when many wish there was a boss who had the authority to order that shift.

Distributed innovation systems like open source will need different organizational means of dealing with that potential trap. Linux developed an interesting mechanism fairly early in its evolution. By 1994 Torvalds recognized that the idea of a developer-user had an inherent, internal contradiction in practice. The developer wants to work on tough new problems and stretch the boundaries of what is possible; the user wants to work with a reasonably stable platform (of course, a single person could be a developer in this sense for part of the day and a user at another time). The answer was a clever organizational innovation—parallel release structures for the code base. The stable code tree is tested, "proven," and optimized for users, while the experimental code tree tries new ideas and experiments with new functionalities. This is a simple idea, but it allows Linux to satisfy two audiences with different and sometimes conflicting needs. And importantly, it also fosters synergy between them because new features are tested in the development tree first and then incorporated into the stable tree as they are proven.

The software lock-in problem is a specific example of a more general phenomenon that was crystallized by James March and later popularized by Clayton Christensen in his book *The Innovator's Dilemma*.[28] Particularly with bounded rationality at play, individuals and organizations alike face a difficult trade-off between the *exploration* of new possibilities and the *exploitation* of old certainties.[29] If you overinvest in exploration, you end up with too many undeveloped new ideas and too few working products. But if you stick with exploitation, refining and improving on the margins of what you already know, you can easily get stuck in a suboptimal trap while someone else creates something truly new. Even in substantively rational models of decision-making, the problem remains because investment in exploration is inherently risky, dependent on developments in other areas that the rational decider cannot control, and often interdependent on the investment decisions of others.

There is no clean solution to this dilemma. Christensen argues that it is the core challenge for successful business strategy; March says, "Maintaining an appropriate balance between exploration and exploitation is a primary factor in system survival and prosperity."[30] Different kinds of organizations create their own distinctive institutions to try to manage the dilemma—businesses have skunk works, academics have sabbaticals and MacArthur fellowships, and so on. Distributed innovation systems like open source have a simple advantage here that helps compensate for the inherent tendency to stick to debugging an existing architecture. Any single open source developer might suffer from individual versions of the innovator's dilemma in her own decision-making about where to invest her time, but the decision as a whole remains disaggregated among the community. Because everyone has access to the source code, anyone can make a different calculation at any time. The story of Samba TNG (Chapter 6) is a good example of this process at work. Ultimately the freedom to fork code distributes much more widely the ability to make different bets on the exploration-exploitation trade-off, and that raises the probability that on aggregate some decisions will pay off. Put simply, end-to-end innovation systems depend on a market logic of disaggregated decisions rather than on a central intelligence to manage the innovator's dilemma.

A second question about end-to-end innovation systems is how well they respond to the diverse needs of users in the many different segments of a complicated community. Placing intelligence at the edges of the network creates demands on users as well as opportunities. Open source software is sometimes criticized as "hackers writing for hackers"—in other words, making too many demands on the kind of

end-user who wants technology to be plug-and-play. Clearly open source software today is stronger on the server side than it is in end-user applications; it is a favorite of sophisticated systems administrators more than the home PC user. The standard explanation for this imbalance is the motivations of the producers—hackers do prefer to write for other hackers. The technical aesthetic that the open source process depends on is more focused on solving complex programming problems than it is on engineering for convenient user interfaces and simplicity. It is true that the mainstreaming of open source products in the early twenty-first century has changed the calculus around that issue to some degree. Market challenges can be seen as technical challenges when they get sufficiently complicated. A truly elegant graphical user interface (GUI) is a technological achievement as well as an achievement of understanding the psychology and practicality of users' needs. Projects like KDE and GNOME (two GUIs for Linux) as well as companies like Red Hat and Eazel (now defunct) take on some of these challenges.

But it is possible that end-to-end innovation systems will generally underperform in these specific kinds of user interface settings, for two related reasons.[31] Some programmers believe that understanding direct user experience requires extensive physical interaction—that is, focus groups, not Internet mailing lists. The more general point here is that design for components like user interfaces still rests on the transmission between individuals of a great deal of tacit information. This requirement makes it difficult to modularize and difficult to develop in parallel distributed settings. End-to-end innovation systems lack a good mechanism for getting at the underlying concepts of tasks like this in novel ways. That may change as advances in information processing move more human experiences out of the tacit realm and into communicable forms, but for now this seems a notable constraint.[32]

A third question is how end-to-end innovation systems interact with standard-setting. Open source processes tend to be powerful magnets that attract standards to form around them. The economic logic (antirivalness, increasing returns, de facto subsidization) is part of the reason. Another part is that open source removes intellectual property barriers that create the possibility of competitive advantage for the owner of the standard. The commercialization of Linux suppliers does

not change that. If Red Hat innovates and that innovation becomes popular in the market, other Linux suppliers can adopt the innovation as quickly as they wish, because they will always have access to the source code and the right to use it.[33]

In other words, releasing source code is a credible commitment against the possibility of using a standard for technological lock-in and hold up. Proprietary standards cannot make this commitment in the same way. Microsoft is often accused of mixing into so-called open standards "secret" APIs, the hooks on which developers build interacting applications.[34] Another accusation against Microsoft is that the company uses its market power to embrace, extend, and extinguish standards—for example, incorporating a standard into Windows but adding extensions to it that increase its functionality in some way but make it incompatible with other, non-Microsoft implementations. This is precisely what the open source community accuses Microsoft of trying to do with Kerberos, a standard Internet authentication protocol that provides the critical service of allowing two remote users on the Internet to verify each other's identity.[35]

These are not arcane technical issues; they have huge consequences for the evolving shape of a network economy. Consider, for example, the web browser war that erupted between Netscape and Microsoft around 1996. If that had extended into a similar war on the server side, two separate World Wide Webs—an Internet Explorer web and a Netscape web—might have been created, with all the resulting costs of duplication and lost synergies. A big reason this scenario did not happen was Apache—the open source product that came to dominate the server side and that could not be hijacked to favor either the Microsoft or Netscape browser. Companies and governments that buy large software packages for long-term use increasingly value this aspect of open source standards because it inoculates them against later exploitation of lock-in by a dominant supplier.

The market power of open source standards may also infect other kinds of software outside its immediate market. The case of Qt is an interesting example. In 1998, Linux developers were building a new graphical desktop interface for the operating system.[36] Matthias Ettrich was behind one of the most promising GUIs called KDE (K Desktop Environment) that was licensed under GPL. The problem was that Ettrich built KDE using a proprietary graphical library called Qt

from a company called Troll Tech. Qt was not fully open source; it could be used freely in some circumstances, but in others Troll Tech required a developer's license at $15,000. Ettrich thought Qt was the best tool for what he wanted to do, and the license allowed him to use it, so he went forward. The promise of an elegant GUI for Linux was sufficiently powerful that some other developers were also willing to look past the problems with Qt licensing or fudge the open source definition in this one case. That temptation met very strong resistance from others in the community, who worried that KDE would succeed and that Linux would then become dependent on nonfree software. At first they tried to convince Troll Tech to change the Qt license; the proposal was that Qt would be GPL code when used in free software but would still be subject to a commercial license if it were used in a commercial product. Troll Tech demurred on the grounds that this would effectively cede control of the code base to the open source community and gut the commercial value of the product.

This looming disagreement set in motion two firm responses from the open source community. The first, led by Miguel de Icaza, was a new GUI project called GNOME that would be built on a fully free graphical library, Gtk+, as an alternative to Qt. The GNOME project immediately attracted a lot of attention and effort from open source developers and notably from Red Hat, which stuck by its strict open source commitment by contributing coders to GNOME.[37] Another set of developers started a project called Harmony, to create a fully open source clone of Qt that could be built into KDE.

As GNOME improved and Harmony demonstrated the promise that it could in fact replace Qt, Troll Tech recognized that Qt risked being surpassed and effectively shut out of the Linux market. In April 1998 Troll Tech set up the KDE Free Qt Foundation, to ensure that a free version of Qt would always be available and to commit itself against later making any claims on Qt-dependent software. Later that year Troll Tech wrote a new license called the Qt public license; in late 2000 the company decided simply to license the code under GPL.

How these dynamics are likely to affect the national and particularly the international politics of standard-setting remains unclear. These politics are typically analyzed around bargaining power between and among firms, national governments, and, to a lesser extent, international institutions. Open source adds an interesting twist. The open

source community is clearly not a firm, though firms represent some of the interests of the community. And the community is not represented by a state, nor do its interests align particularly with those of any state. The community was international from the start and remains so; its motivations, economic foundations, and social and political structures bear very little obligation to national governments. We simply do not know how this community will interact with formal standards processes that remain deeply embedded in national and international politics as well as industry dynamics. (There are some relevant and researchable examples—The Linux Standard Base, Internet Engineering Task Force (IETF), and to some extent the Internet Corporation for Assigned Names and Numbers (ICANN)—but that is a subject for another book.) I will return to the question of how national governments may interact on this issue and whether it might be possible to manipulate the dynamic in ways that would yield national advantage.

My fourth point returns to the upside through a relatively narrow perspective—what might end-to-end innovation in software mean for the foundations of the information economy? Quite simply, it could be revolutionary, possibly more so than the rapid expansion of hardware capabilities has been over the last twenty years. As I said earlier, there is no Moore's Law for software. But software is the rate-limiting factor of the information economy and, as hardware capabilities continue to advance, the mismatch becomes increasingly evident.[38] Whether or not open source software will replace Windows on the desktop is irrelevant from this broader technological perspective. At a higher level of abstraction, the question becomes, Can open source software become an enabler of a next generation of information-processing and communications tools, by accelerating the rate of performance improvement in the rate-limiting step?

There are suggestions that it is already having this effect as we move toward ubiquitous computing, the presence of vast information-processing capabilities in everything humans use, from simple household appliances to distributed supercomputers. At the bottom end of this spectrum, open source programs have major advantages: They are highly configurable, cheap, and most important, available for experimentation without restrictions. An inventor who wants to build an intelligent toaster has free access to the source code of an operating sys-

tem or other programs without royalties or licensing schemes. The home video recorder Tivo runs on an open source operating system built from Linux. IBM and Nokia among others are experimenting with open source operating systems for the next generation of mobile phones and communications devices.[39] In 2000 AOL announced a project to build a mass market Internet appliance on the Transmeta Crusoe chip running a scaled-down Linux and a simple open source browser. There are many such projects; what is significant in this example is that there is no Intel processor and no Microsoft software. At the higher end, the impending move to widespread use of 64-bit processors (Intel's Itanium and AMD's Hammer) puts Linux and Windows essentially on equal starting ground in the competition with Sun's 64-bit Ultrasparc servers that run Solaris (Sun's proprietary Unix).

The most interesting possibilities for acceleration are in the use of open source software in distributed or cluster computing. At the very high performance end of the computing spectrum where supercomputers build simulations of extraordinarily complex phenomena like climate patterns, the often custom-designed hardware is extraordinarily expensive and system software stands out clearly as the weakest link.[40] As early as 1994, researchers began experimenting with open source software tools to tie together large numbers of standard, off-the-shelf personal computers into a de facto supercomputer. This is called Beowulf architecture. In 1998 researchers at Los Alamos built a Beowulf-class supercomputer from 140 Alpha microprocessors running Red Hat Linux. It ran at 47.7 gigaflops, which made it number 113 in the list of the world's 500 fastest computers, and cost $300,000—about one-sixth the cost of a 48-gigaflop supercomputer from Silicon Graphics.[41] IBM's Los Lobos project in 2000 tied together 512 Pentium 3 processors running Linux and achieved 375 gigaflops, making it the world's twenty-fourth fastest supercomputer. In 2001 IBM built similar machines for the U.S. Department of Defense, the National Center for Supercomputing Applications, Royal Dutch/Shell, and others. In 2003 open source cluster computing moved to the forefront of technology for massive multiplayer online gaming, which (despite its recreational purpose) is an extremely demanding set of applications.[42]

These and many other developments in cluster computing are important not just because they reduce the price of supercomputing.

The more significant change is that cluster architecture begins to move high-performance computing into the realm of a commodity—something that can extend to lots of places and can advance much more quickly than single-application, highly customized systems. This could be as disruptive to economic and social practice as was the widespread dissemination of the desktop PC in the late 1980s. The next step, analogous to how the Internet tied together personal computers, is the move toward grid computing, whereby anyone with a simple connection to the next-generation Internet could access the mass of processing power that is distributed throughout the network.[43] Software again is the rate-limiting factor to grid computing. The emerging standard comes from the Globus project, an open source collaboration between national labs, industry, and universities.

It is easy to dream up science fiction–style scenarios about how the grid will bring supercomputing power to anyone, anywhere, and transform the way humans think, communicate, and solve problems. I don't need to indulge in that here simply to point out that the next generation of end-to-end innovation will depend on lots of software code, reliable code, but most importantly compatible code, recombinant code that interfaces straightforwardly with other code. The open source process has shown that it is possible to build this kind of code within its own end-to-end, distributed innovation setting. The next chapter in the story of distributed innovation will probably be an order of magnitude more surprising in its revolutionary implications, and make the consequences of the first-generation Internet seem quaint.

The Commons in Economic and Social Life

More than thirty-five years ago Garrett Hardin introduced the metaphor "tragedy of the commons" into common parlance.[44] He argued that resources that are rival and held in common (for example, a grazing pasture) tend to be depleted by overuse because each user passes some of the costs of use onto others. If the resources cannot be depleted (for example, a digitally encoded symphony) but need to be provided by voluntary human action in the first place, a related tragedy can develop if nonexcludability means that no one has an incentive to provide the commons good to start. Diverse theoretical and em-

pirical critiques of Hardin's argument have chipped away at the edges, but the core insight remains powerful and central to law and economics debates about intellectual property in the digital era. Technology has brought this debate to a crossroads. It is now, or soon will be, possible efficiently to "privatize" or "enclose" all intellectual products. CDs can be copy-protected; books can have embedded chips that charge you every time you copy a page, or go back and read a page more than once; and a law like the DMCA makes it illegal to build tools that circumvent these systems regardless of how those tools are used.

It is not hard to make an economic efficiency argument that would support this kind of regime. After all, when I spend $48 for a book, I am actually buying a package of rights to use that book in a variety of ways, only some of which I might really want (for example, I might want to read only a few chapters, photocopy a particular chart, and quote one paragraph). Disaggregate these uses, charge separately for each, and a deadweight loss of efficiency can be extruded from the package. At this very stark level, there is no conceptual room left for an intellectual commons. Fair use of copyrighted materials, on this logic, exists today only because the transaction costs of making a diverse set of contracts for the limited use of the material in different ways would be prohibitively high.[45]

But this version of the efficiency argument is too monochromatic. Legal scholars have argued eloquently for the particular importance of a distinct commons in communications and intellectual property as a whole.[46] These arguments often invoke the open source phenomenon as an example of how core common infrastructures can be provided and maintained in ways that contribute to welfare while remaining outside the boundaries of what James Boyle called the "second enclosure movement."[47]

Open source code indeed mimics a commons along lines of an early FSF insight: nondiscrimination matters more than market price *per se*. The distinctiveness of the commons lies in freedom to "use" the commons without restriction, and forever. Embedded here are at least two discrete claims about the functions of a commons and why it plays a valuable, even essential role in economic and social life. These claims need to be separated cleanly so they can be developed, evaluated, and aligned against the narrow efficiency story, now that technology has made it a realistic alternative.

The first and most familiar argument is that a commons functions as a feedstock for economic innovation and creative activity. Put simply, this is the stuff that you can use to "build on the shoulders of giants" without having to ask their permission or pay them a restrictive license fee. I am not here going to take on the huge literature debating the parameters around that argument as well as the practicalities of how it could and should be implemented. I want simply to separate out and emphasize the economic logic behind the core insight, which leads to a commons producing an increase in innovation and aggregate welfare (instead of a tragedy). Think of a commons simply as a set of inputs to a production process. Now imagine privatizing some of those inputs. This would change their price and availability on the market, acting much like a tariff does in trade theory. This then changes the behavior of producers, who make substitutions. Choices about what inputs to use become biased systematically toward inputs that the producer itself owns or can easily assemble the rights to use (in other words, those without the tariff). In a world of perfect information and costless transactions, people could trade these rights and scramble their way back toward an optimal allocation of resources. In this world, the costs of inputs go up, availability becomes selectively biased, and aggregate welfare suffers.

Heller and Eisenberg call this "the tragedy of the anticommons" to distinguish it from the routine underuse that is a part of any functioning intellectual property rights system (and is justified as a bargain for incentivizing production in the first place).[48] This is not just theory; it is an increasingly visible problem—for example, in pharmacogenomics research in which complex and overlapping patent claims on genetic code make it difficult to assemble the package of rights needed to produce a drug. Benkler makes a convincing case that creative industries would tend to substitute inputs from intrafirm sources and owned inventory because they can use those inputs at marginal cost, while they would have to pay additional fees (as well as pay the extra costs of searching for and assembling rights) to the owners of other information inputs.[49] This story is clearly and powerfully applicable to software code as well.[50]

Recognize that in none of these cases is it really possible to quantify the deadweight loss of an anticommons tragedy. More importantly, it is not possible to precisely compare those costs to the incentives that

might be lost (or gained) by moving the parameters of intellectual property protection. Each term in the equation is uncertain, which means that the arguments tend to corrode into rhetoric or theology.[51] Prescriptive arguments about what the law should do depend heavily on counterfactuals that are plausible but no more than that. I am not saying that public policy can be made without these kinds of arguments; clearly it is all we have to go on. I am pointing out that the innovation and aggregate welfare argument for the commons will not by itself secure its intellectual foundation against the challenge of technological change.

There is a second argument about the function of a commons that is less frequently made explicit. This argument points to the importance of participation in creative activity by individuals, simply for the sake of participation itself. In other words, creative action is an individual good regardless of whether it contributes to measurable aggregate creativity, innovation, or social welfare. We think it is important and valuable that children draw pictures of clouds even though not a single one is as good as those drawn by Van Gogh. We value the explosion of individual creativity facilitated by the web even though the vast majority of what is "published" there adds little that is measurable to society's stock of knowledge. And we value these things for their own sake, not because they can be interpreted as investments in an educational process that may later yield unspecified innovation.

These may sound like fuzzy notions in the middle of what feels primarily like an economic discourse; but they of course have a long pedigree in many different fields of thought, from developmental psychology to theories of participatory democracy, and increasingly within studies of human-computer interaction.[52] And so it is important to point out how the open source software process fits within this argument. Consider the case in favor of constitutionally protected speech, which (apart from its social value as an error-correcting mechanism) lies in part with the enhanced autonomy it confers on individuals. Speech demands a nondiscriminatory infrastructure. Software code is not precisely speech, but it is coming close in some sense to being the infrastructure for communication. Certain bodies of code are essential tools of expression, in the same way that pens and paper were for an earlier technological era. It enhances personal autonomy to own those tools, or at least to be sure that no one else owns them in a restrictive

way. Would it be good for democracy if newsprint were engineered to dissolve when someone tried to photocopy it? Or if you could only write on it in particular languages?

Consider also the value of how open source facilitates competence development. I am not saying that people using Linux will become skilled programmers or that they should. I am saying that systems administrators, students, hobbyists, and the occasional end-user will be incentivized, enticed, or tempted to develop a slightly higher level of competence with their tools. At a minimum they will have the opportunity to use knowledge and resources they have at hand to try to solve a problem, a kind of personal efficacy that many people feel technology has progressively taken away from them. This too may sound romantic. But the nature of the relationship between individuals and their tools of production—a core component of political identity—does change with technology and custom, and it is not locked in by human nature. Let me go out on a limb and suggest that those who see hints of a new class ideology developing around information technology are not necessarily wild-eyed.[53] "Bit-twiddlers" are neither exactly proletariat nor bourgeoisie. They may not own the means of production in the sense that Marx argued, but they certainly do have significant control over those means, in a more profound way than the term "symbols analysts" or "knowledge workers" captures.[54] As a rough generalization, they value science and technological problem-solving elegance equally at least with profit. Technocracy rather than a hierarchy of money is their route to authority and respect. Is this the kind of "class" that Piore and Sabel, for example, believed might emerge from the second industrial divide, emerging instead from the divide between industrial and information era production? It is far too soon to say, and the prospect at least for some is far too attractive not to be skeptical of premature claims, so I simply leave this as a question.

These claims about the social and economic importance of the commons are not to say that Hardin was mistaken—rather, that he saw only part of the picture. And I am not making the argument that current intellectual property thought is "wrong." It is not wrong, but neither is it right. It simply modulates incentives to produce and distribute things in particular ways. A great deal of critical thinking has gone into sketching out the downside implications of extending the realm, scope, and time frame of exclusive intellectual property protections to

digital goods. The open source process and the analytics of the commons that follows from it help to illuminate some of the upside implications of a different property rights regime. Neither regime necessarily drives out the other; they have coexisted for some time and will probably continue to do so, even though the boundaries may shift.[55] Move the boundaries and some things will not be produced. Some incentives will unravel. Something will be lost. On the other hand, something will be gained as well. It is harder in some sense to label and name those gains because we are still in the midst of developing a language to talk about them and a set of arguments to categorize and evaluate them. Discussing in explicit terms the value of a commons, including but also outside of its contributions as a feedstock for innovation, is an important part of that process.

Development and International Economic Geography

The digital divide between developed and developing countries is now a central feature of international politics and the global economy. The slogan captures a fundamental disparity in access to and the ability to use new technologies, a reflection of long-standing divides of poverty, education, and freedom to make choices. What impact might the open source process have on this set of issues?

The question is embedded in a defensible worry that digital technologies may be set to exacerbate global inequality. In fact it is easy to write a lock-out scenario in which developing economies risk falling further behind the leading edge of a digitizing world economy. The combination of Moore's Law (rapid increases in processing power at declining prices) and Metcalfe's Law (positive network externalities, meaning that the value of the network increases disproportionately as it grows) suggests that markets can grow intensively and dramatically *within* the developed world, without necessarily having to expand geographically at the same pace. As developed economies create networked purchasing and production systems that depend on advanced digital technologies, countries that are not connected on favorable terms (and firms within those countries) are deeply disadvantaged. International organizations and nongovernmental organizations are increasingly computer-enabled as well, which means they will favor interaction with countries and organizations in the developing world that

are similarly enabled and can interact effectively with their information systems.

The point is that sophisticated information technology capabilities are becoming a prerequisite to effective interaction with the world economy. And while the prerequisites have grown, so have the potential downsides of lacking them. The industrial economy may have had inherent limits to growth, implying that exclusion of much of the world's population was actually necessary in some sense (it is impossible to imagine a world in which every family in China burns gasoline in an internal combustion engine). There are no such inherent limits to the information economy that we can now see. From an efficiency perspective, the possible exclusion of 4 billion people from the next era of wealth creation makes no sense. From an ethical standpoint, it is even more problematic than was the exclusion of previous eras, because there is no intrinsic environmental or resource-based reason for it.

In practice a lockout of emerging markets from the global information economy is unlikely. For developing countries that do participate, the key issue will be the terms of interconnection, much as were terms of trade in the post–World War II global economy. That experience provides cautionary tales. Dependency theorists in the 1970s identified perverse patterns of development whereby some emerging economies were incorporated into multinational production networks primarily for extractive purposes, either via exploitation of raw materials or low-cost labor. In either case there was insufficient productive investment and little potential for the emerging economy to upgrade its position over time. The resulting patterns of development were skewed definitively in favor of the developed world, while the dependent economies suffered deteriorating terms of trade and increased vulnerability to business cycle fluctuations. Or they found themselves with "enclaves" that were connected to the global economy but almost completely separate from the local economy, offering few benefits to the developing country as a whole. In both cases, the hoped-for second-order development effects spreading to the broader economy were limited at best. Contemporary arguments about "body-shopping" for low value–added data entry and simple programming in India, Malaysia, and other developing countries reflect a logic that is very similar to the insights of dependency theory.

If you accept the proposition that software is an enabler of the next phase of economic growth and development, it makes sense to overlay what we know about the open source process onto this kind of story and put forward some hypotheses about how it might change. Start with the simple notion that software is a tool for manipulating information. If the tool is essentially free to anyone who wants to use it, and freely modifiable to make it useful in whatever way the user can manage, then lots of people will grab the tool and experiment with it. The argument about end-to-end architecture is just as valid across countries as it is within countries. The open source community has been international from the start, and it remains so. That is more than simply noting that open source developers live all over the world. It is important that developers in China, Indonesia, and other developing countries contribute to open source software; but what is more important is that they all have access to the tool, and on equal terms. The open source community transcends national boundaries in a profound way because its interests (as well as its product) are not tied to or dependent on any government.

The degree to which a software tool can be used and expanded is limited in practice. But with open source software, it is limited only by the knowledge and learning of the potential users, not by exclusionary property rights, prices, or the power of rich countries and corporations. I remain cautious in thinking about what this means: Knowledge and learning are real constraints, but they are a different kind of constraint than are exclusionary rights and power. The free diffusion of tools will not create a profound leveling phenomenon. Even when everyone has equal access to tools, some people can and will use those tools to create more value than others. Consider an analogy to an imaginary world in which everyone had access to as many steam engines as they wanted, all at the same time, at nearly no cost, and with an open ability to disassemble, customize, and reassemble the components. Economic development would still have been uneven, but it might have been *less drastically uneven* than it is today.

If extrapolating this thought experiment to the information economy sounds outlandish even as a hypothesis, take a step backward toward more familiar discussions within developmental economics about the question of "appropriate technology" for poor countries. For most of the second half of the twentieth century, rich country governments and international development institutions were the ones making the most important decisions about "appropriate technology." Open source software shifts the decision-making prerogative into the hands of people in the developing countries. In one sense the provision of a freely available technological infrastructure represents by itself a form of wealth transfer to poor countries, but it is a wealth transfer that developing countries can maneuver to their particular advantage. To provide real products and services on top of the infrastructure requires an investment of local labor. India, China, and many other developing countries have a surplus of inexpensive technical manpower. Combining this with free software tools creates the possibility of an interesting kind of competitive advantage that would certainly matter in local markets and in some cases might be important in global markets as well.[56] One of the advantages of the GPL is that it then prevents a dysfunctional enclosure of mobilized "southern" resources into "northern" properties protected by patents that are offered for resale to the "south" at exploitative prices, a depressingly common pattern for knowledge-intensive products as diverse as music, plant varieties, and pharmaceuticals.

The potential to invigorate information productivity in developing economies is real. One of the consequences of the extraordinarily rapid growth of raw processing power over the last decades is that leading-edge users and the innovative applications that they develop for their own practical purposes have actually been the drivers and shapers of technological change, because they are the creators of meaningful demand for better, faster, and cheaper computing. It's well known that military demand played a major role in the first generations of computer technology. Census bureaus, banks, and insurance companies drove a second generation through the use of huge databases. The web drove innovation in the 1990s. It is likely that the process of annotating and processing the information released in the human genome project will be an important driver of the next generation of innovation.

Recognize that each of these "lead-use" applications came from the developed world, principally from the United States. Some years ago development theory posited that emerging economies were simply less sophisticated and less advanced versions of developed economies and that they would transit the same stages to arrive at essentially the

same place some years later. This so-called stage theory is now discredited. Emerging markets have their own autonomous development logics. This will be just as true for economies transiting the information revolution as it was for those transiting the industrial revolution. The promise inherent in this argument is that software innovations can and should come from everywhere. Emerging markets are not implicitly stuck relying on commoditized, hand-me-down innovation from the developed world. They can have their own lead users who pull technology development toward applications that fit specifically the indigenous needs and demands of emerging markets. Indeed, because information technology is more easily customized than were many industrial-era technologies, the opportunity for autonomous lead users in emerging markets to deeply influence the direction of technology development is considerable. Open source software helps to tap this potential. For indigenous demand to be expressed, users really have to understand the menu of possibilities they face and the ways in which a digital infrastructure could contribute to their lives. When those possibilities are evolving as quickly as they are today, it seems certain that users generate demand primarily through a process of learning by doing.[57] Only over time and with increasing familiarity do users gradually come to understand and then to imagine what the technologies can do for them. The empowerment that comes with free access to source code is not then simply a gratifying emotional experience; it is a necessary economic prerequisite of evolving demand.

In many cases the "killer apps" for developing economies, which are the applications that find widespread acceptance and drive technology forward, will almost certainly come from *within* those economies. In other words, many design principles developed in the United States for American users are not going to be directly transferable to the developing world. The same is true of the granular peculiarities of payment systems and even more so of e-governance applications. The Simputer project, a $250 portable computer (running Linux) designed and built by a consortium from the Indian Institute of Science and the Bangalore-based software company Encore, is an illustrative example of how this process can work. This remarkable machine provides connectivity to the Internet, allows sharing among multiple users, supports text-to-speech capability for illiterate persons, and provides voice feedback in local languages.[58] In other words, it is a leading-edge experiment in a mass-market information appliance, not a cheapened or dumbed-down version of the Palm Pilot.

It is also the case that many countries have distinct political and security incentives to avoid lock-in to proprietary software products. Cities in Brazil have been at the forefront of legislation mandating government offices to use free software when possible; similar laws have been proposed in France, Argentina, Italy, Spain, and others. In spring 2002 a Peruvian congressman defended a similar bill brought up in Lima with these arguments: "To guarantee national security or the security of the State, it is indispensable to be able to rely on systems without elements which allow control from a distance or the undesired transmission of information to third parties. Systems with source code freely accessible to the public are required to allow their inspection by the State itself, by the citizens, and by a large number of independent experts throughout the world. Our proposal brings further security, since the knowledge of the source code will eliminate the growing number of programs with 'spy code.'"[59]

I quote at length because the letter demonstrates vividly that the issue is more than cost-savings. There is nationalist ideology here but also concrete interests. It is no surprise to industrial organization theorists that governments (like any customer) want to avoid locking themselves into a single private provider for crucial tools. And it is no surprise to international relations theorists that states want to avoid becoming dependent on software whose export is under U.S. legal jurisdiction and whose development is controlled by America's dominant software industry. Communications networks, e-government applications, and of course just about everything that makes up a modern military force increasingly run on sophisticated software. No national government, if it had alternatives, would have chosen during the twentieth century to accept dependence for steel or petroleum on a single supplier or a small number of suppliers based in a potential rival nation. And so it is no surprise that the Chinese government in particular has supported the development of Red Flag Linux and other open source packages as a distinct alternative to proprietary software—in part as a development tool, and in part as a lever to reduce potential dependence on a company that just happens to be based in Redmond, Washington, USA.[60] The emerging global security environment after September 11, 2001, will likely accelerate these trends, as the United

States capitalizes on its advantages in information processing to contribute to its new doctrine of preemptive security.

Of course information technology and open source in particular is not a silver bullet for long-standing development issues; nothing is. But the transformative potential of computing does create new opportunities to make progress on development problems that have been intransigent. In the broadest sense, the potential leverage on development comes not from software itself, but from the broad organizational changes that the open source process as a way of making software will drive. I am not arguing that developing countries can use the open source process to make up for lack of sufficient legal and economic infrastructure, or replace institutions by installing Internet connections. I am saying that there are interesting possibilities for building systems of distributed innovation within emerging economies that lead to autonomous innovation. This could have a significant impact on development prospects.

Power, Transaction Costs, and Community

In 1968 two of the most influential figures behind Arpanet argued that computer networks would have radical transformative effects on human thought and society.[61] Particularly since the Internet became popularized in the late 1990s, there has been an outpouring of literature on "virtual communities," the social life of people and information within networks, as well as organizational and business implications. There is far too much rich imagination, critical theory, and (increasingly) empirics within this literature for me to summarize it here. My point is to lay out a core analytic that has close ties to a governance argument in the similarly unstructured space of international politics, overlay the open source process on top of that analytic, and explore some implications for thinking about the governance of communities. The core analytic is made up of two parts, relational power and transaction costs.

The concept of relational power is simply that power is an attribute of relationships rather than of actors *per se.* Bundles of capabilities do not by themselves give anyone or any organization the capacity to get others to do what they otherwise would not do, or set the terms for cooperation and conflict. Power instead takes shape within complex rela-

tionships in which bargaining happens on many different levels, and lots of stuff (both material and symbolic) is exchanged.

Modern theories of international politics grappled with relational power under the heading of "complex interdependence."[62] A key puzzle that needed to be solved in that setting was to explain the development of stable cooperative relationships, "regimes" made up of norms, principles, rules, and decision-making procedures that were (surprisingly) durable despite their nonauthoritative basis.[63] Regime theory built on the fundamental insight of the Coase theorem to propose a solution. With firm property rights and sufficiently low transaction costs so the actors involved could readily make contracts, markets could handle externalities reasonably well—in fact, at the limit, markets would produce an efficient outcome without any need for authoritative regulation. As long as the international system remained anarchic, or lacking an authoritative governance structure capable of enforcing contracts, this seemed a vital mechanism for sustaining decentralized cooperation between autonomous states. The trick was to find a way to bring down the relatively high transaction costs that seemed to characterize much of international politics. That then was the core function of international regimes—to reduce transaction costs so international politics as a whole could move somewhat closer to a Coase-style equilibrium.[64]

A great deal of recent thought about e-commerce and Internet community tracks this logic of relational power and transaction costs. The ability to move information around the world without friction has been deeply associated with a market metaphor, even more deeply a market-based ontology as a way of seeing the world. Two related things point in this direction: decisions being pushed down either to the individual or to the machine on a case-by-case basis, and the massive reduction of transaction costs enabling those individuals (or machines) to find each other and agree to an exchange. And so the Internet has often been portrayed as a "perfecter" of markets, bringing a vision of efficiency ordered through "perfect" information (as economists say) and Coasian equilibrium arrived at in relationships outside of authority.

This story is most clearly expressed in the business literature, of course. But it is also an important part of the stories that people tell about affinity communities facilitated by communications technology.

The worldwide community of stamp collectors existed long before the Internet, but the Internet has made it possible to "fix" many of the "market failures" that the community must have suffered (for example, imperfect price and availability information for particular artifacts, individuals in remote places who did not trade as much as they wanted to, and so on). People suffering from rare diseases presumably would have liked to communicate with each other to exchange information, support, and so on before the Internet, but the transaction costs of finding each other around the world were prohibitively high.

In this kind of discourse, the failure of a community to take shape because of high transaction costs is just as much a market failure as is the nonevent of a contract to exchange money for widgets that would make both sides better off but does not happen because the widget owner can't find the buyer. The perfection of markets and the realization of potential communities are theoretically identical.

Let me take this argument one step further before I break it open. *Wired* magazine's former editor Kevin Kelly talks about a vision of a world made up of "smart" objects—a world in which everything has an embedded microprocessor and its own Internet address. He once excitedly explained, "This chair I'm sitting in will have a price in real-time, if you want to sit in it you will know exactly how many resources you will have to trade to do that."[65]

That would be efficient. But multiply that kind of interaction many times over into a social system and ask yourself if it is the kind of world you want to live in. There is a more profound point that I want to make explicit, about the way in which the metaphor and the accepted ontology of markets constrains how people think about the possibilities that technology is spawning.

Imagine a smart chair, connected to a lot of other smart things, with huge bandwidth between them, bringing transaction costs effectively to zero. Now ask yourself, With all that processing power and all that connectivity, why would a smart chair (or a smart car or a smart person) choose to exchange information with other smart things through the incredibly small aperture of a "price"? A price is a single, monolayered piece of data that carries extraordinarily little information in and of itself. (Of course it is a compressed form of lots of other information, but an awful lot is lost in the process of compression.) The question for the perfect market that I've envisioned above is, Why

compress? My point is that even a perfect market exchange is an incredibly thin kind of interaction, whether it happens between chairs or between people, whether it is an exchange of goods, ideas, or political bargains. I want to emphasize that communities, regimes, and other public spheres can come in many different shapes and forms. The "marketized" version is only one, and it is in many ways an extraordinarily narrow one that barely makes use of the technology at hand.

One way to open up this story is to go back and reexamine its roots in the Coase theorem. Changes in transactions costs (like any other costs) are phenomena that happen at the margin (in the economic not pejorative sense). But of course low transaction costs are only one ingredient in the Coase equilibrium. The other is secure, well-defined property rights; and as I have said, these property rights are now in play in a new way. My argument here simply is that shifting property rights can and will likely destabilize the foundations of existing cooperative arrangements and institutions, and possibly in more radical ways than do changing transaction costs. According to Coase, the transition to a new set of "stable" property rights would be the trickiest part for institutions to navigate. And it seems almost certain to me that we will be living through such a transition for at least the next decade.

One important aspect of institutional innovation happening within that transition can be seen within new manifestations of networks. More specifically, a variety of networks are emerging around different answers to two questions that lie at the core of governance (even in a network): What are the major resources of power and control, and what are the ordering principles? Consider a schema that captures a slice of that variety (see Figure 5).[66] The axes are self-explanatory and are meant simply to capture three distinct kinds of variance in network interactions. Connectivity is the technological enabler that pushes experimentation with all three kinds of networks out on their distinctive trajectories. And three very different kinds of networks emerge.

eBay is an example of a massive, low-transaction-cost, one-to-one electronic flea market where buyers and sellers are empowered to find each other and contract through a modified auction process. eBay itself provides the trading platform and a reputation management system that facilitates single-shot interactions between nearly anonymous parties.[67] Power in this self-described "community" lies with the provider of the trading platform, because eBay itself writes the rules for

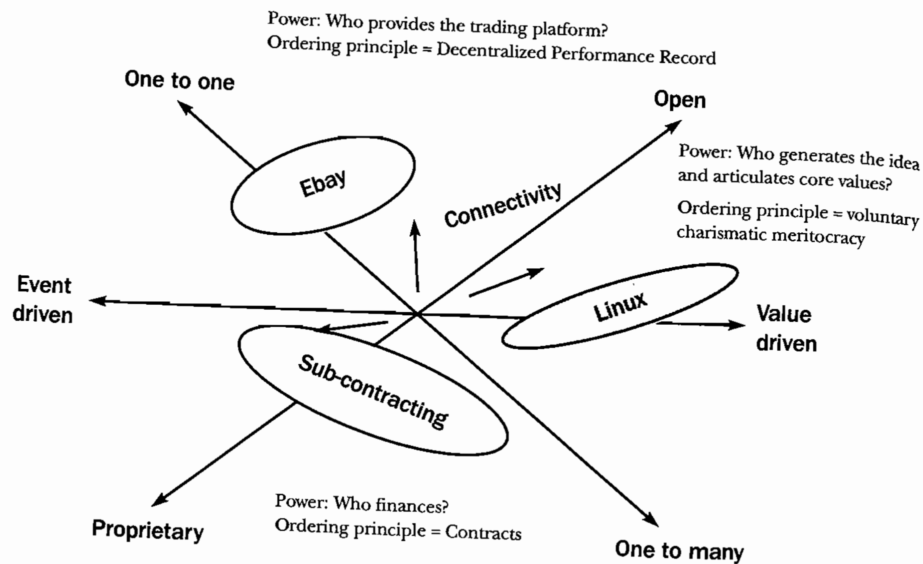

Figure 5 Varieties of networks.

transactions and the operation of the reputation system. Governance is decentralized to the participants and focused on assuring the performance over time of the trader, rather than the quality of the underlying asset that is being traded.

The subcontracting or Hollywood model is a very different kind of network. The organizing principle is simply a highly disaggregated or outsourced division of labor. If I want to make a movie, I search for and assemble the specific inputs I need—a director, some actors, a lighting designer, a sound producer, and many more—around the task of creating the film. These inputs will work together for several months and then (just as quickly as I brought them together) disband, to later reassemble with other inputs in some new configuration for another movie. This is the exemplar of a virtual organization, but there is really nothing particularly virtual about it, other than the fact that technology reduces the costs of identifying and assembling the optimal inputs for my particular project. Power in this community is closely tied to finance, because it is the investment that acts as glue to pull together a temporary coalition of skilled inputs. The individual or organization that invests in bringing together the virtual organization

also holds the reins of governance because that player writes the contracts that set the terms of interaction.

The open source model occupies the third niche in this scheme. The community is formally as open as eBay, and more open in the sense that opportunities for voice are greater. But rather than a single-shot contract or even a series of them, the community is connected by a shared goal of creating a common product. Like the Hollywood model, collaboration requires an individual to hold many (not just one) other individuals in working relationships. Power in this community, at least to start, belongs to the person who generates the idea and articulates the core values behind the project. But as the community takes shape, its very openness moves power away from the leader and toward the followers, as I discussed. Governance remains voluntary throughout, in large part because of this odd trajectory of power. A rough meritocracy coexists with charismatic personalities, a charisma that has the distinctive characteristics that I explained in Chapter 6.

Connectivity will drive each of these kinds of networks further along their respective trajectories.[68] The eBay model will move toward an increasingly perfect market, through more efficient management of reputations, better price discovery, more liquidity. The Hollywood model will move toward more elaborate specialization, extend its geographical scope, and so on. One of the things the open source model will do is to show the limits of some basic notions about the functions of large-scale governance. Principal-agent problems are not the big "problem" that governance mechanisms have to solve in this setting. As I explained, distributed innovation on the open source model does not depend on bringing agents' incentives into line with those of principals. The distinction itself looks arbitrary—who exactly is the principal and who is the agent in this environment? Another and related thing the open source model will do is challenge more broadly some of the tenets of how we think about pluralism within organizations, and how pluralism scales. Many organizations, including some international organizations, today claim a special normative status because of their pluralist composition.[69] The challenge is that the distinctiveness is going away. Pluralism at many different levels is being enabled by communications technologies and by experimentation with property; together, these are reducing the marginal cost of adding voices toward an asymptote of zero. Open source demonstrates how it is then possi-

ble for an increasingly large number of actors, public and private, to enter the contest for control over the channels by which technical expertise (or claims of expertise) flow. Neither international nor any other politics is heading all the way to one big pluralist society in which anyone can be part of any organization. But the default position is, indeed, changing—as the active choice becomes a matter of whom to *exclude* rather than whom to *include*. As more inclusive and pluralist organizations grow up in the space of international politics and economics, organizations like today's international organizations will come to look less special and the legitimacy they have drawn from that special status will dissipate. I think about this ongoing process as a kind of "disaggregation" of legitimacy, and I think the spreading of claims on legitimacy to a variety of unexpected and unlikely actors is set to continue, in international politics as much as elsewhere.[70]

Networks, Hierarchies, and the Interface

As Secretary of State in the Nixon Administration, Henry Kissinger famously asked, "When I call Europe who answers the phone?" Behind this glib comment lies a profound theoretical issue and a very practical set of questions for governments, corporations, and other organizations. What Kissinger really was asking is, How does a hierarchically structured government (the United States) deal effectively (communicate, cooperate, or compete with) a powerful institution structured in a fundamentally different way? Take this one step of generalization further and the question becomes what are the dynamics of increasingly dense relationships between hierarchies and networks.

Microsoft is asking parallel questions about open source. In 2001 Microsoft essentially declared war on open source software.[71] Remember that Microsoft (for better or worse) is one of the most profoundly successful organizations on earth when it comes to strategy. It has an extraordinarily well-honed system for managing its relationships with other corporations (too successful by some accounts). It is just as expert and nearly as successful in managing its relationships with governments. But Microsoft has no strategy template for managing a relationship, even a hostile one, with the open source community. You can't buy that community; you can't drive it out of business; you can't hire away the talent; and you can't really tie it up in the courts (although

Microsoft has tried each of these tactics).[72] Imagine someone asking, "When Microsoft calls Linux who answers the phone?" The question makes no sense. It presupposes a structural configuration of an organization that is not true of many networks and certainly not of the open source community.[73]

National governments face similar problems. For many years the United States has been fighting an undeclared war against terrorist networks like al-Qaeda. After September 11 the U.S. government made that a de facto declared war. But modern war is a social convention that evolved between national governments. One state declares war on another and the other declares war back. Each makes demands on the other; their ambassadors meet; they try to negotiate an end to violence; they bargain and fight in some evolving mix and then perhaps sign a treaty. It is a gruesome repertoire, but both sides understand what their roles are and how the game is played. The war against terrorism is different. A national government, perhaps the largest and most hierarchical national government on earth, has declared war on a transnational network. That network has certainly committed acts that we call war against the United States, but it has made no explicit demands on the United States. In fact it does not even announce itself as the attacker. How do you bargain with an enemy that hides and doesn't tell you what it wants? Who do you bargain with? It is a comforting fiction, but still a fiction, that Osama Bin Laden is the political equivalent of a president or king. Loosely coupled cell-like network structures act sometimes in coordination and sometimes not.

The underlying analytic that portrays war as a bargaining game between two discrete and similarly structured actors that differ primarily in terms of power and preferences was a convenience for international relations scholarship that helped people think about large-scale international conflict during the Cold War. But more important differences now lie in how organizations are structured and how those structures complicate setting the terms of their interaction, even in war. Complex networks have at least five characteristics that complicate how they relate to hierarchies:[74]

- Structural complexity: The "wiring diagram" can be extraordinarily intricate.
- Network evolution: The wiring diagram can change quickly.

- Connection diversity: The links between nodes vary in strength, direction, and even sign of influence (+ or -, excitatory or inhibitory).
- Dynamical complexity: The state of each node can vary rapidly.
- Node diversity: There are many different kinds of nodes.

And finally there is the notion of meta-complication: Each of these complexities can influence and exacerbate each other.

There is a huge literature evolving across the social sciences looking at the structure and functioning of networks. I focus here on the related but different question, What happens at the interface, between networks and hierarchies, where they meet? The interface between differently structured systems is typically a very creative place where new forms of order, organization, and even life arise. In physics it has been called the edge of chaos, where order meets disorder and where phase changes take place. In evolutionary theory, it is a source of new species, not just marginal variation. In organization theory, boundary spanners are a major creative force behind new kinds of organizations.[75] This is also the place where the relationship between the open source process and more traditional forms of organization for production are being worked out. The general point is that one of the key social science challenges at present is to conceptualize more clearly how hierarchically structured organizations (like large governments and corporations) develop and manage relationships with network organizations. I have found no good comprehensive model, but there are analytic suggestions about the kinds of problems arising at the interface that need to be solved. One set of problems involves communication. Information flows among differently structured channels in hierarchies and networks. It may be encoded in different protocols, perhaps even what amount to different languages. Another set of problems falls around coordination. A de facto division of labor between the network and the hierarchy (even in a conflictual relationship) needs a coordination mechanism, but neither price nor authority will cross the interface successfully.

A literature in business and foreign policy as well as a more theory-oriented literature in international politics is beginning to grapple with these problems. The first two seem to take their cue from the important arguments of sociologists like Dimaggio and Powell, who developed a powerful argument about isomorphism, detailing some of the pressures driving organizations that are connected to each other in highly dense relationships to change so they come to look more like each other structurally.[76] And so it is common to read how formerly hierarchical organizations, particularly large businesses, are in the process of transforming themselves into networks. In the foreign policy and security field, David Rondfeldt and John Arquilla have for almost ten years been making prescient arguments about the rise of networks in international conflict and the implications for what they call "netwar."[77] Their policy propositions—"hierarchies have a difficult time fighting networks, it takes networks to fight networks, and whoever masters the network form first and best will gain major advantages"—track the institutional isomorphism literature by encouraging hierarchical governments to remake their security organizations as networks to interface successfully with their networked adversaries.

These arguments and the policy strategies they carry will likely prove themselves quite useful in the longer term. But in the medium term, isomorphic pressures on institutions are just that—pressures, not outcomes. It is important to resist the fiction that national governments and large corporations are all going to become networked organizations in the foreseeable future, because they won't. And terrorist organizations are not soon going to become hierarchical structures with clear lines of command, ambassadors, and physical capitals. The reality is more complicated: Both forms will coexist and have to find ways to relate to each other.

In international relations theory, Margaret Keck and Kathryn Sikkink have asked similar questions about relationships between transnational "advocacy networks" and national governments.[78] Other studies of global social movements talk about the emergence of "complex multilateralism" to describe a form of governance that emerges in the interaction between international organizations and transnational networks.[79] These are valuable perspectives so far as they go. But they still suffer from an unfortunate "bracketing" of the hierarchal structures as that which is somehow "real" or concrete, while trying to prove that networks "matter" vis-à-vis more traditional structures.[80] The next question they naturally ask, "Under what conditions do networks matter?" is premature unless the answer can be well structured in terms of necessary and sufficient conditions. To get to that point requires a

good conceptual articulation of the space in which the game of influence is being played out. And that is still lacking.

The open source story as I have told it here points to a different way forward. Two distinct but equally real organizational forms exist in parallel to each other. The dynamic relationship between hierarchies and networks over time determines both the nature of the transition and the endpoint. One form may defeat the other through competition. Both may coexist by settling into nearly separate niches where they are particularly advantaged. Most interesting will be the new forms of organization that emerge to manage the interface between them, and the process by which those boundary spanners influence the internal structure and function of the networks and the hierarchies that they link together.

If my generic point about creativity at the interface is correct, it is then my strong presumption that this is a problem suited for inductive theorizing through comparative case study research. The war against terrorism, the relationship between open source and proprietary models of software production, and the politics among transnational NGO networks and international organizations share characteristics that make them diverse cases of a similarly structured political space. I am certain that some of the most interesting processes in international politics and economics over the next decade are going to take place in this space, at the interface between hierarchies and networks (rather than solely within either one). Comparing what evolves in diverse instantiations of that space is one way forward.[81]

Generalizing Open Source

The success of open source is a story about software. If it were only that, it would still be important for social scientists thinking about cooperation problems. And it would still have significant implications for economic growth and development.[82] That is the minimum case. If the open source process has more general characteristics, if it is a generic production process for knowledge that can and will spread beyond software *per se*, then the implications might be considerably larger. My purpose here is not to take a definitive position on this question, but rather to lay out some of the conditions that embed it.

Markets and hierarchies are reasonably good at coordinating some

kinds of human behavior and not very good at others. Creative intellectual effort, for example, is highly individuated, variable in its transparency, often tough to measure objectively, and thus very difficult to specify in contracts. Another way to make the same point is to say that firms and markets only tap into a piece of human motivation—an important piece certainly, but for many individuals only a small part of what makes them create. This should sound obvious to anyone who is or has worked with artists, musicians, craftspeople, authors, dancers, and so on. There is nothing unique to the Information Age here.

Many of these creative endeavors are practiced by single individuals or by small groups. When you put them together into a large group, the dynamics of cooperation change. Orchestras have conductors, who struggle with finding a balance between individual creativity and the joint product that a symphony represents. Companies that rely heavily on knowledge and creativity spend enormous effort trying to strike a similar balance. Leaders of organizations like these will often say they feel like they are tapping into 25 or 30 percent of what their employees have within them, and if only they could find a way to access some of that unused 70 percent. This, I think, is the reason why the open source phenomenon attracts such curiosity outside the software world. If this experiment in organization has found a way to tap a greater percentage of human creative motivation (if only 10 percent extra), then the question of how to generalize and expand the scope of that experiment becomes a very interesting one to a much broader group of people.

One important direction in which the open source experiment points is toward moving beyond the discussion of transaction costs as a key determinant of institutional design. Make no mistake: Transaction costs are important. The elegant analytics of transaction cost economics do very interesting work in explaining how divisions of labor evolve through outsourcing of particular functions (the decision to buy rather than make something). But the open source process adds another element. The notion of open-sourcing as a strategic organizational decision can be seen as an efficiency choice around distributed innovation, just as outsourcing was an efficiency choice around transaction costs.

The simple logic of open-sourcing would be a choice to pursue ad hoc distributed development of solutions for a problem that (1) exists

within an organization, (2) is likely to exist elsewhere as well, and (3) is not the key source of competitive advantage or differentiation for the organization. Consider, for example, financial companies that, when merging with other financial companies, need to build a software bridge that links different types of legacy databases and computer systems. This is a notoriously difficult problem that financial companies typically outsource to specialist consulting firms. That decision creates a useful division of labor for the financial company in question, but it is extremely inefficient for the industry as a whole, which has to start almost from scratch and recreate solutions to this kind of problem every time a major merger goes through.

An alternative strategy would be to open source a solution to the problem. Dresdner Kleinwort Wasserstein did something like this with the help of Collab.net to stimulate the creation of a developer community around a base of source code. The idea was to find people in other places sharing a similar problem, involve them in a common developer effort, spread the development cost around the organizations, and increase the mobilized talent pool. If measuring the parameters were easy, the strategic choice to open source would be a function of comparing benefits you gain from the distributed development effort to costs you incur getting it started along with savings you will bring to your competitors by giving them free access to the same tools that you have. Because no one can measure those parameters in advance (and often not even after the fact), the decision becomes an experiment in organizational innovation. We are seeing many such experiments now and will likely see more of them as the success of open source becomes more widely understood. But there is no inherent reason this experiment will be bounded within information technology.

Consider this scenario. A petrochemicals company owns the intellectual property rights to a specific refining technique that it has developed at considerable cost in its own labs. But because the problem that this technique solves is generic and well known, an international network of graduate students is collaborating in an open source–style process to try to solve the same problem. The open source process is advancing rapidly and generating a lot of attention. Some of the engineers who work for the petrochemical company are contributing to the open source process in their spare time—it seems to be a more creative endeavor through which they can play with ideas more freely than they can at work. At some point, the engineers and the executives at the company realize that the open source process has produced a refining technique as good as or better than their proprietary technique. And now they face a major decision. They could try to fight the open source alternative in the market or in the courts. They could try to hire the graduate students who built the code and attempt to enclose the knowledge within their organization. Or they could embrace the new process and encourage their engineers to engage with it so they could understand it better and customize it to the company's advantage. They could build the plants that use the new process, improve it as they learn by doing, manage the production, marketing, and regulatory issues for the product, and so on.[83]

I think it likely that this scenario will come to pass in various economic sectors and that there will be experiments in managing it from each of these angles (and others). One of the next steps in research on open source should be to build analytic models that try to specify conditions that favor or hinder these experiments. I do not have a full-fledged model to propose at this point, but I suspect any model will have to focus on at least two factors that transaction cost economics does not emphasize. The first is the relationship between problem-solving innovation and the location of tacit information, information that is not easily translated into communicable form.[84] As information about what users want and need to do becomes more fine-grained, individually differentiated, and hard to communicate, the incentives grow to shift the locus of innovation closer to them by empowering them with freely modifiable tools. The second is the importance of principles and values, in contrast to efficiency as an organizing concept. This goes back to the example of the smart chairs and the eBay-style community. Increasingly efficient economic exchange has its own decreasing marginal returns, both as an organizing principle and as an explanatory variable for looking at Linux-style communities and the conditions under which they will form.

A note of caution: As open source has begun to attract broad public attention over the last few years, the term itself has been overused as a metaphor. There are now experiments with an open-cola alternative to Coke and Pepsi, an "openmusic" registry, an "openlaw" project at Harvard Law School, and any number of "open content" projects to build mass encyclopedias, for example. Many of these are simply "open" fo-

rums in the sense that anyone can contribute anything they wish to a mass database. Others are essentially barn-raising volunteer projects that look for interested people around the world to make a donation to a common pool. Many of these projects gain their ideological inspiration from the open source process and tap into some of the same motivations. Many are interesting experiments in using Internet technologies to organize volunteer efforts and affinity groups. But in many instances these projects are not organized around the property regime that makes the open source process distinctive. That is not any kind of criticism; it is simply an identification of a difference, but it is an important difference that needs attention at this early stage of building models that specify conditions under which the open source process will be favored.

Experimenting with open source production in different knowledge domains will involve a lot of learning by doing. Consider, for example, the structure of medical knowledge in a common family practice type setting, which is interestingly parallel to the structure of knowledge for in-house software development. My doctor in Berkeley has a hard problem to solve. I present myself to her with an atypical sinus infection, low-grade fever, aching muscles, and a family history of heart disease. The bad news is that I represent to her a highly customized configuration and a finely grained problem. The good news is that there almost certainly is a similarly configured patient presenting somewhere else at the same time. At the very least, other doctors are solving pieces of my problem in other settings.

The second piece of bad news, though, is that she will find it extremely difficult to access that distributed knowledge and thus will most likely have to figure out my problem without much help. In fact doctors have very cumbersome means of upgrading the common medical knowledge that they draw on to support their work (it takes a long time to publish papers in journals; these papers are not configured for easy recombination; and so most doctors facing an unfamiliar problem will call friends to see whether they have faced a similar problem recently, or know someone who has). Certainly bandwidth has been a limiting factor in the effective use of knowledge (a typical CAT scan picture is a huge data file). But as the bandwidth comes on line, the key issue will be the social organization of knowledge-sharing and upgrading. From the perspective of the medical care system as a whole,

there is enormous wasted effort when the doctor in New York ends up going through a problem-solving process that a doctor in Berkeley has already figured out in a very similar case, simply because she has no way to search for and access that knowledge efficiently—even though the California doctor has no reason not to share it (and could very much benefit from diffuse reciprocity in a future reverse "trade"). In software development the open source process is a way of sharing precisely information of this character in an effective way. As I have explained it in this book, the process depends on culture, technology, organization, and to some extent the nature of the knowledge that is being shared.

Primary care medicine has similar cultural characteristics and a parallel organization of practice; what seems to be missing is the intersection between the technology and the nature of the knowledge to be shared. Many doctors would today claim that much problem-solving knowledge in medicine cannot really be put into communicable form. An X-ray picture surely is knowledge that can be digitized, while the skill of how to make a diagnosis by palpating a patient's chest is a tacit form of knowledge that each practitioner has to recreate for himself. But is the difference with software source code really a difference of kind? At one time source code was also difficult to communicate—before the Internet, people had to carry large tapes and disks around from place to place. When bandwidth increases, the parameters change. The distinction between tacit and digitizable knowledge may be less severe than practitioners in any particular field of endeavor tend to think. After all, source code is not easy to read; there is tacit knowledge embedded in the structure of an elegant solution to a programming problem that experts have and novices don't. Would a massive expansion of bandwidth show that "tacit" knowledge in medical practice has similar characteristics?[85] I pose that as an honest, not rhetorical, question, with the caveat that a positive answer would make certain aspects of medical practice an obvious place to experiment with open source style knowledge production.

A second and in some sense obvious application of the open source process might be in genomics.[86] The human genome sequence published by the National Institutes of Health and Celera in 2001 is at best in fact a rough draft. The next decade (at least) of genomic science will be spent correcting and "annotating" the sequence, separating sig-

nal from noise, interpreting what information is being used by human cells and in what ways, essentially debugging an enormous body of code.

This is the most ambitious and probably the most important knowledge production task of the first half of the new century. It is increasingly taking place *in silico* (on a computer) rather than *in vitro* (in a test-tube). The barriers to entry are getting lower as the DNA sequencing and manipulation technologies become cheap and available, which is the same kind of transition that happened to computing with the creation of the PC. Ten thousand dollars, a fast Internet connection, and a couple graduate courses in biochemistry are all you need to become a "bio-hacker." And much of the basic source code to hack is freely available (in both senses of the word "free"). Viable economic structures could be built up around open source genomics. For example, the farmer might not buy prepackaged seeds from a distributor. Instead he would buy a conveniently packaged genetic sequence, which he would then be free to modify for his own particular setup and conditions, and then synthesize ("compile") the DNA into a functioning genome for his corn crop. He might offer his sequence customizations to others under a license like the GPL, while companies spring up to provide auxiliary services for a fee (such as helping farmers to organize the crop options that would be available to them and make them work together well on a particular piece of land).

There are at least two important cross-cutting pressures against this kind of evolution. The culture of the genomics community is simply not as open as is the culture of the computer science community. The same underlying norms of sharing that software engineers built up in the early years of computing and are part of the history of that community are not available for the genomics community in the same way, at least not in its recent history. Rather than recapturing a version of the Unix-style past for the basis of a different narrative, the genomics community would probably have to import these ideas about alternative ways of organizing from outside, or be "invaded" by a new set of players who carry those ideas. The second pressure is external, the force of government regulation. Government action facilitated the development of the open source software process (even if not intentionally). In contrast, the liberal patent rules for genetic data allowed a much more restrictive intellectual property regime to evolve quite

early around genomics. September 11 and its aftermath—particularly the doctrine of preemptive security applied to bioterrorism—reinforces that regime. It is far too early to tell how these conflicting pressures will balance out, but there is no doubt they form an interesting (if not very well controlled) experiment for generalizing open source.

The open source community itself has been circumspect about trying to generalize a set of conditions under which the model could spread. Some of this caution is tactical—Eric Raymond has said more than once that he wants to see open source win the "battle of ideas" in the software world and consolidate its home base before too much energy gets expended anywhere else. But the software world is not likely unique, and successful models of production do not always behave in politically expedient ways. The arguments in this book point to a set of general hypotheses that I have presented over the course of the narrative and analysis, and I summarize them here in conclusion. They fall into two broad categories, the nature of the task and the motivations of the agents.

The open source process is more likely to work effectively in tasks that have these characteristics:

- Disaggregated contributions can be derived from knowledge that is accessible under clear, nondiscriminatory conditions, not proprietary or locked up.
- The product is perceived as important and valuable to a critical mass of users.
- The product benefits from widespread peer attention and review, and can improve through creative challenge and error correction (that is, the rate of error correction exceeds the rate of error introduction).
- There are strong positive network effects to use of the product.
- An individual or a small group can take the lead and generate a substantive core that promises to evolve into something truly useful.
- A voluntary community of iterated interaction can develop around the process of building the product.

Some of these hypotheses shade off into arguments about the motivations and capabilities of the agents. To be more precise, the open

source process is likely to work effectively when agents have these characteristics:

- Potential contributors can judge with relative ease the viability of the evolving product.
- The agents have the information they need to make an informed bet that contributed efforts will actually generate a joint good, not simply be dissipated.
- The agents are driven by motives beyond simple economic gain and have a "shadow of the future" for rewards (symbolic and otherwise) that is not extremely short.
- The agents learn by doing and gain personally valuable knowledge in the process.
- Agents hold a positive normative or ethical valence toward the process.

These are expansive hypotheses; the parameters in many instances would be hard to measure and specify *a priori.* As is the case for many such hypotheses about social processes, they are broadly indicative of the kinds of conditions that matter. They tell you where to look, and even more so where not to look, for answers. That may not be fully satisfying, but it's not a bad place to start when you are looking at something so intriguing and at the same time unfamiliar.

NOTES

INDEX