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 imperfect, *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 projects 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,

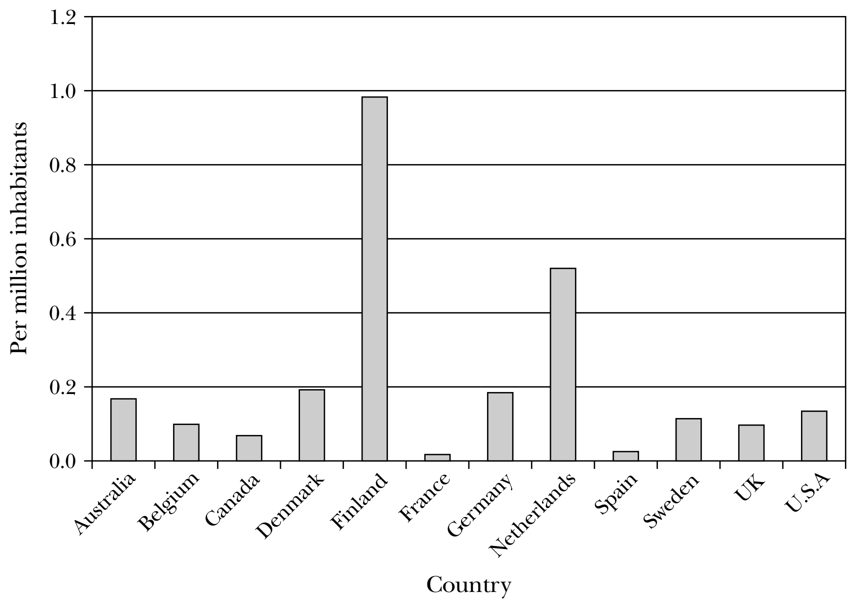

Figure 1 Linux code authors listed in first credits file (1994), concentration by country.

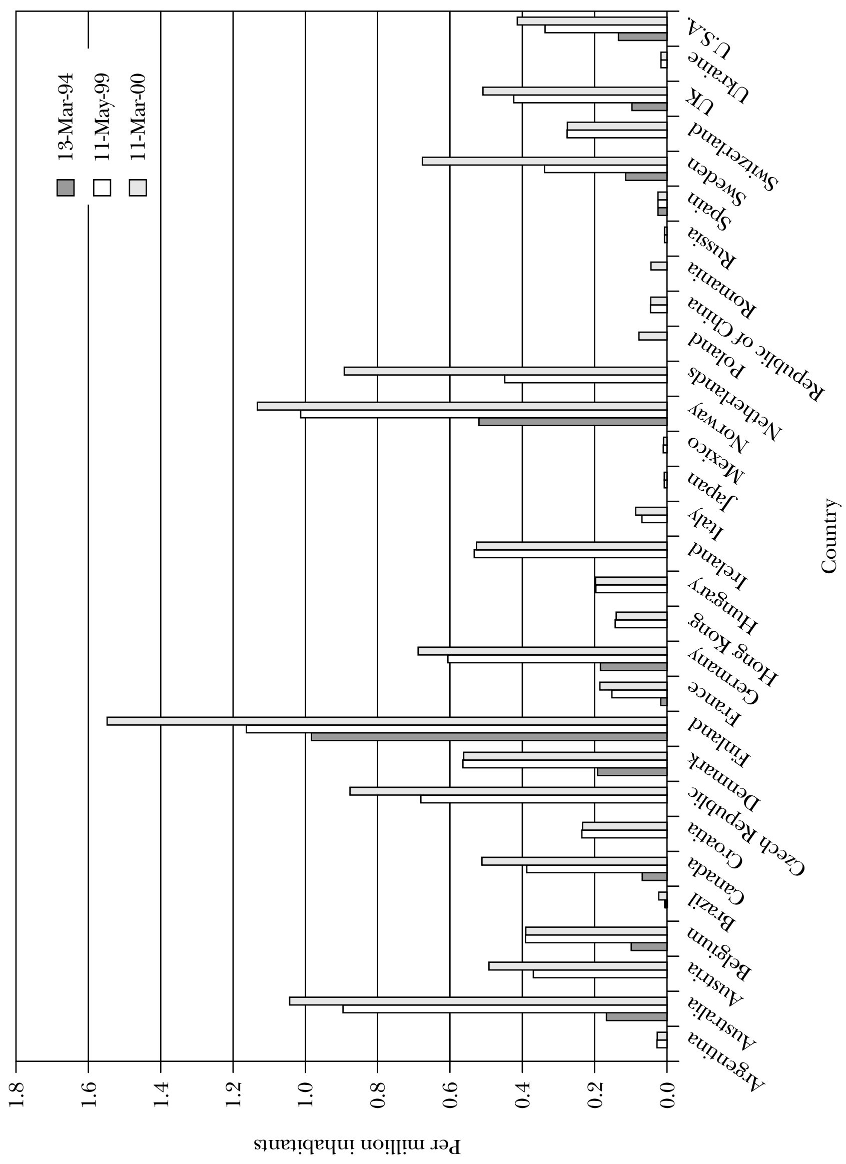

Figure 2 Linux code authors listed in credits files, concentration over time.

.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

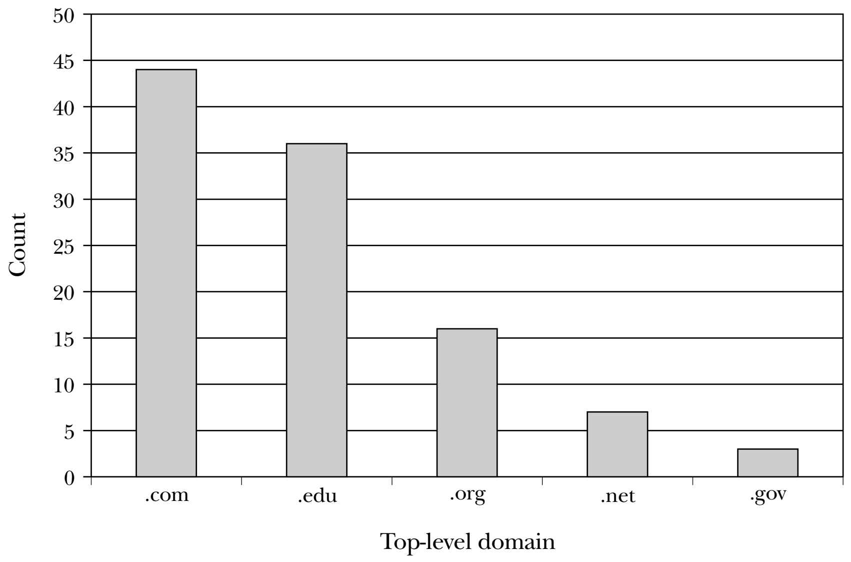

Figure 3 Linux code authors in 2.2.14 credits file, by top-level domain of email address.

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 individ-

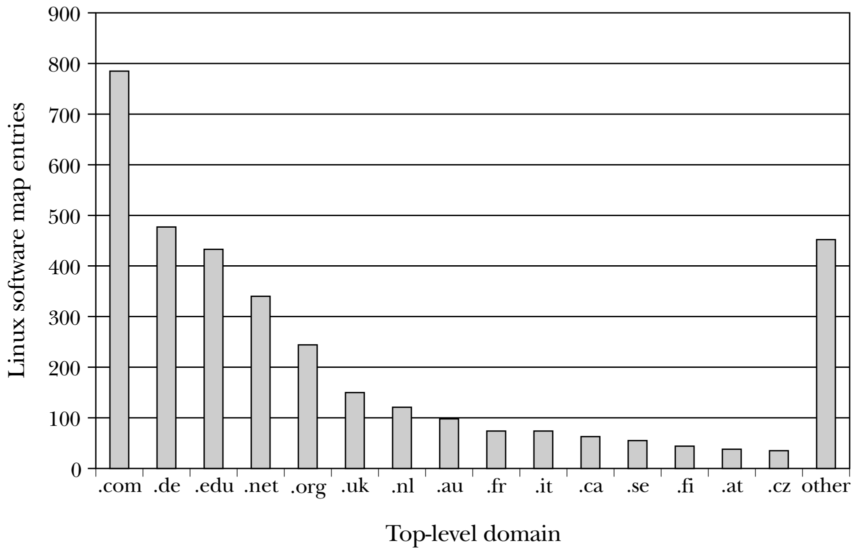

Figure 4 Linux software map (LSM) entries, by top-level domain of email address.

ual'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]

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 complain 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 organization 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-deprecating. 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 different 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.